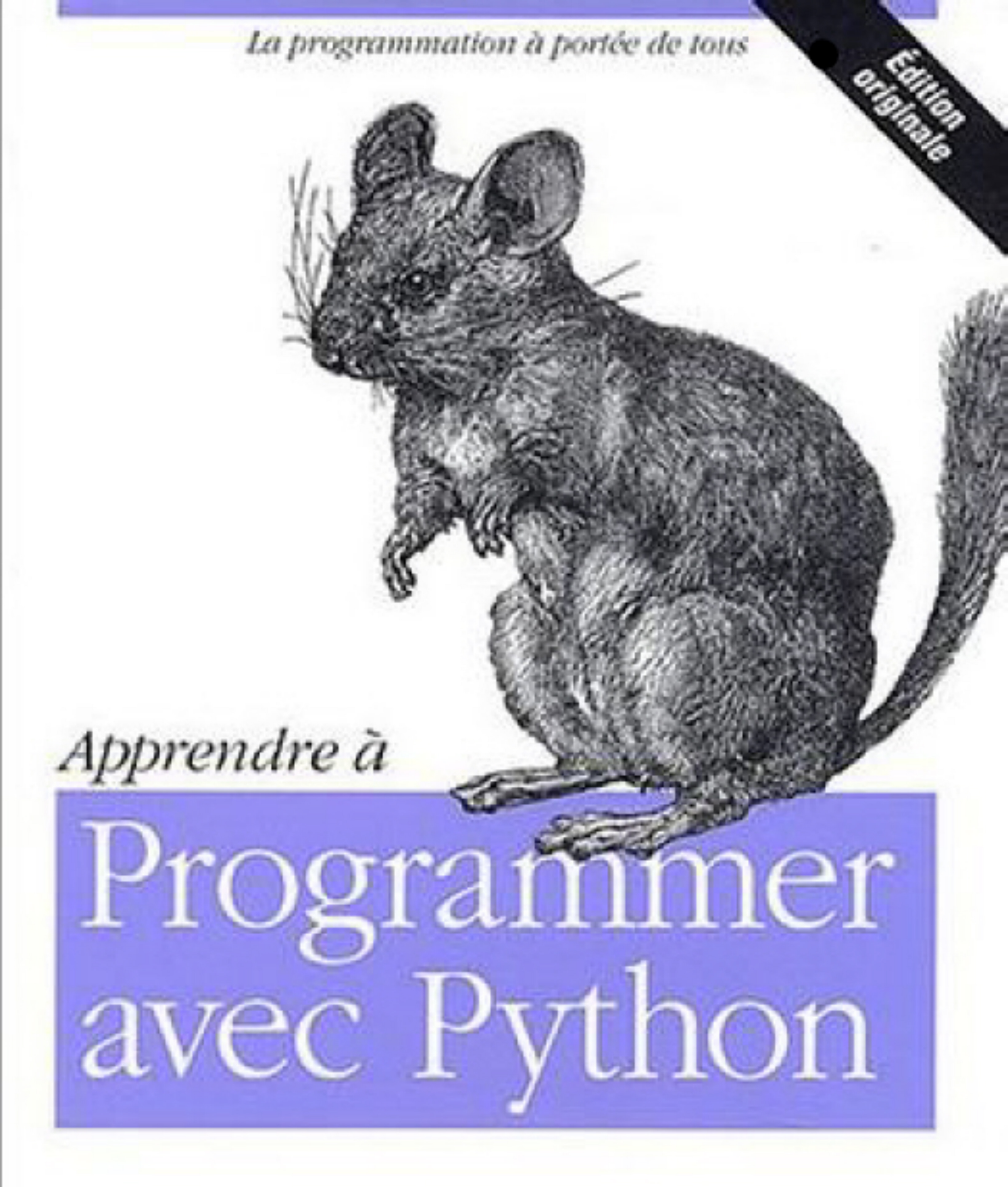
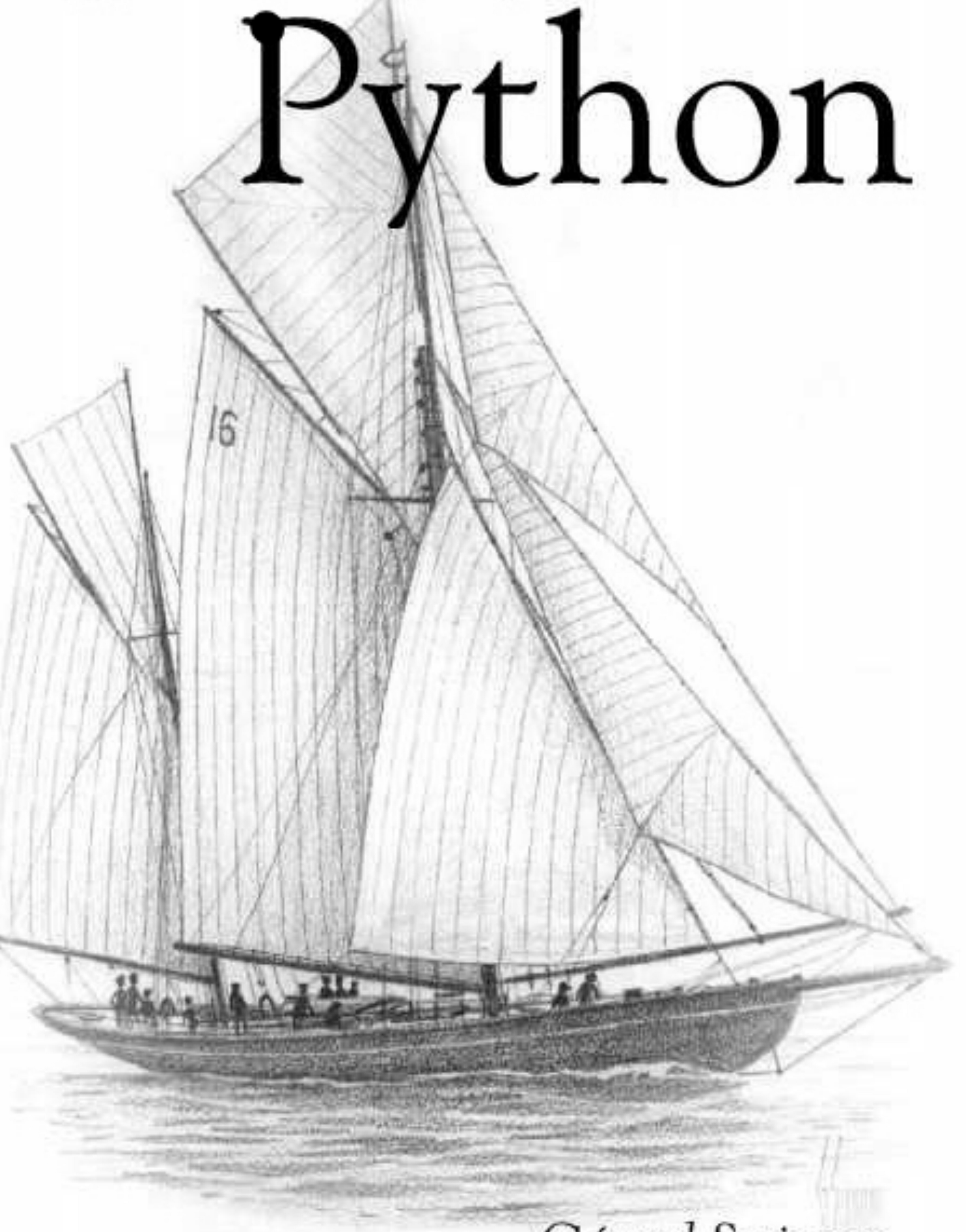


O'REILLY®

Gérard Swinnen

Apprendre à programmer avec

Python



Gérard Swinnen

Adaptation libre de "How to think like a computer scientist"
de Allen B. Downey, Jeffrey Elkner & Chris Meyers

Colophon

Choisie délibérément hors propos, l'illustration de couverture est un dessin réalisé par l'auteur à la mine de graphite sur papier Canson en 1987, d'après une photographie ancienne. Il représente le yacht de course de 106 tonnes *Valdora* participant à une régate dans la rade de Cowes en 1923.

Construit vingt ans plus tôt, et d'abord gréé en yawl, *Valdora* remporta plusieurs trophées avant d'être regréé en ketch en 1912 avec la voilure de 516 m² que l'on voit sur le dessin.

Ce superbe voilier, très estimé par ses équipages pour son bon comportement à la mer, a navigué pendant près d'un demi-siècle.

Apprendre à programmer avec Python

Notes de cours

par **Gérard Swinnen**
professeur et conseiller pédagogique
Institut S[t] Jean Berchmans - S[te] Marie
59, rue des Wallons - B4000 Liège

Ces notes peuvent être téléchargées librement depuis le site :
http://www.ulg.ac.be/cifen/inforef/swi

Une part importante est adaptée du texte :
How to think like a computer scientist
de Allen B. Downey, Jeffrey Elkner & Chris Meyers
disponible sur : http://rocky.wellesley.edu/downrey/ost
ou : http://www.ibiblio.org/obp

Introduction

Les présentes notes ont été rédigées à l'intention des élèves qui suivent le cours *Programmation et langages* de l'option *Sciences & informatique* au 3[e] degré de transition de l'enseignement secondaire belge. Il s'agit d'un texte expérimental qui s'inspire largement de plusieurs autres documents publiés sous licence libre sur l'*internet*.

Nous proposons dans ces notes une démarche d'apprentissage non linéaire qui est très certainement critiquable. Nous sommes conscients qu'elle apparaîtra un peu chaotique aux yeux de certains puristes, mais nous l'avons voulue ainsi parce que nous sommes convaincus qu'il existe de nombreuses manières d'apprendre (pas seulement la programmation, d'ailleurs), et qu'il faut accepter d'emblée ce fait établi que des individus différents n'assimilent pas les mêmes concepts dans le même ordre. Nous avons donc cherché avant tout à susciter l'intérêt et ouvrir un maximum de portes, en nous efforçant tout de même de respecter les principes directeurs suivants :

- L'apprentissage que nous visons doit être adapté au niveau de compréhension et aux connaissances générales d'un élève moyen. Nous nous refusons d'élaborer un cours qui soit réservé à une "élite" de petits génies.
- Dans cette option d'études et à ce niveau, l'apprentissage doit rester généraliste : il doit mettre en évidence les invariants de la programmation et de l'informatique, sans se laisser entraîner vers une spécialisation quelconque.
- Les outils utilisés au cours de l'apprentissage doivent être modernes et performants, mais il faut aussi que l'élève puisse se les procurer en toute légalité à très bas prix pour son usage personnel. Toute notre démarche d'apprentissage repose en effet sur l'idée que l'élève devra très tôt mettre en chantier des réalisations personnelles qu'il pourra développer à sa guise.
- L'élève qui apprend doit pouvoir rapidement réaliser de petites applications graphiques.
  Les étudiants auxquels on s'adresse sont en effet fort jeunes (en théorie, ils sont à peine arrivés à l'âge ou l'on commence à pouvoir faire des abstractions). Dans ce cours, nous avons pris le parti d'aborder très tôt la programmation d'une interface graphique, avant même d'avoir présenté l'ensemble des structures de données disponibles, parce que nous observons que les jeunes qui arrivent aujourd'hui dans nos classes "baignent" déjà dans une culture informatique à base de fenêtres et autres objets graphiques interactifs. S'ils choisissent d'apprendre la programmation, ils sont forcément impatients de créer par eux-mêmes des applications (peut-être très simples) où l'aspect graphique est déjà bien présent. Nous avons donc choisi cette approche un peu inhabituelle afin de permettre à nos élèves de se lancer très tôt dans de petits projets personnels attrayants par lesquels ils puissent se sentir valorisés. Nous leur imposerons cependant de réaliser leurs projets sans faire appel à l'un ou l'autre de ces environnements de programmation sophistiqués qui écrivent automatiquement de nombreuses lignes de code, parce que nous ne voulons pas non plus masquer la complexité sous-jacente.
- Dans notre démarche, nous souhaitons aussi familiariser les étudiants le plus tôt possible avec le concept informatique d'objet, approché par étapes successives. Nous leur ferons d'abord utiliser en abondance divers types d'objets préexistants (et notamment des objets graphiques), afin qu'ils apprennent à exploiter petit à petit les méthodes et attributs de ces objets. La construction d'objets personnalisés sera envisagée plus tard, et progressivement, lorsque nous serons assurés que les notions de base sont déjà bien en place.

Choix d'un premier langage de programmation

Il existe un très grand nombre de langages de programmation, chacun avec ses avantages et ses inconvénients. L'idéal serait certainement d'en utiliser plusieurs, et nous ne pouvons qu'encourager les professeurs à présenter de temps à autre quelques exemples tirés de langages différents. Il faut cependant bien admettre que nous devons avant tout viser l'acquisition de bases solides, et que le temps dont nous disposons est limité. Dans cette optique, il nous semble raisonnable de n'utiliser d'abord qu'un seul langage, au moins pendant la première année d'études.

Mais quel langage allons-nous choisir pour commencer ?

Lorsque nous avons commencé à réfléchir à cette question, durant notre préparation d'un curriculum pour la nouvelle option Sciences & Informatique, nous avions personnellement accumulé une assez longue expérience de la programmation sous *Visual Basic* (*Micro$oft*) et sous *Clarion* (*Top$peed*). Nous avions également expérimenté quelque peu sous *Delphi* (*Borl@nd*).
Il était donc naturel que nous pensions d'abord exploiter l'un ou l'autre de ces langages (avec une nette préférence pour *Clarion,* qui reste malheureusement peu connu).

Si nous souhaitons les utiliser comme outils de base pour un apprentissage général de la programmation, ces langages présentent toutefois deux gros inconvénients :

- Ils sont liés à des environnements de programmation (c.à.d. des logiciels) propriétaires.
  Cela signifie donc, non seulement que l'institution scolaire désireuse de les utiliser devrait acheter une licence de ces logiciels pour chaque poste de travail (ce qui risque de se révéler assez coûteux), mais surtout que les élèves souhaitant utiliser leurs compétences de programmation ailleurs qu'à l'école seraient implicitement forcés d'en acquérir eux aussi des licences, ce que nous ne pouvons pas accepter.
- Ce sont des langages spécifiquement liés au seul système d'exploitation *Window$*. Ils ne sont pas "portables" sur d'autres systèmes (*Un!x*, *M@c*, etc.). Cela ne cadre pas avec notre projet pédagogique qui ambitionne d'inculquer une formation générale (et donc diversifiée) dans laquelle les invariants de l'informatique seraient autant que possible mis en évidence.

Nous avons alors décidé d'examiner l'offre alternative, c.à.d. celle qui est proposée gratuitement dans la mouvance de l'informatique libre[1]. Ce que nous avons trouvé nous a enthousiasmés : non seulement il existe dans le monde de l'*Open Source* des interpréteurs et des compilateurs gratuits pour toute une série de langages, mais le véritable cadeau consiste dans le fait que ces langages sont modernes, performants, portables (c.à.d. utilisables sur différents systèmes d'exploitation tels que *Window$*, *Linux*, *M@cOS* ...), et fort bien documentés.

Le langage dominant y est sans conteste *C/C++*. Ce langage s'impose comme une référence absolue, et tout informaticien sérieux doit s'y frotter tôt ou tard. Il est malheureusement très rébarbatif et compliqué, trop proche de la machine. Sa syntaxe est peu lisible et fort contraignante. La mise au point d'un gros logiciel écrit en *C/C++* est longue et pénible. (Les mêmes remarques valent aussi dans une large mesure pour le langage *Java*).

1 Un logiciel libre (*Free Software*) est avant tout un logiciel dont le code source est accessible à tous (*Open source*). Souvent gratuit (ou presque), copiable et modifiable librement au gré de son acquéreur, il est généralement le produit de la collaboration bénévole de centaines de développeurs enthousiastes dispersés dans le monde entier. Son code source étant "épluché" par de très nombreux spécialistes (étudiants et professeurs universitaires), un logiciel libre se caractérise la plupart du temps par un très haut niveau de qualité technique. Le plus célèbre des logiciels libres est le système d'exploitation ***GNU/Linux***, dont la popularité ne cesse de s'accroître de jour en jour.

D'autre part, la pratique moderne de ce langage fait abondamment appel à des générateurs d'applications et autres outils d'assistance très élaborés tels *C++Builder*, *Kdevelop*, etc. Ces environnements de programmation peuvent certainement se révéler très efficaces entre les mains de programmeurs expérimentés, mais ils proposent d'emblée beaucoup trop d'outils complexes, et ils présupposent de la part de l'utilisateur des connaissances qu'un débutant ne maîtrise évidemment pas encore. Ce seront donc au yeux de celui-ci de véritables "usines à gaz" qui risquent de lui masquer les mécanismes de base du langage lui-même. Nous laisserons donc le C/C++ pour plus tard.

Pour nos débuts dans l'étude de la programmation, il nous semble préférable d'utiliser un langage de plus haut niveau, moins contraignant, à la syntaxe plus lisible. Veuillez aussi consulter à ce sujet la préface de *"How to think like a computer scientist"*, par Jeffrey Elkner (voir page 273).

Après avoir successivement examiné et expérimenté quelque peu les langages ***Perl*** et ***Tcl/Tk*** , nous avons finalement décidé d'adopter ***Python***, langage très moderne à la popularité grandissante.

Présentation du langage Python, par Stéfane Fermigier[2].

Python est un langage portable, dynamique, extensible, gratuit, qui permet (sans l'imposer) une approche modulaire et orientée objet de la programmation. Python est developpé depuis 1989 par Guido van Rossum et de nombreux contributeurs bénévoles.

Caractéristiques du langage

Détaillons un peu les principales caractéristiques de Python, plus précisément, du langage et de ses deux implantations actuelles:

- Python est **portable**, non seulement sur les différentes variantes d'*UNiX*, mais aussi sur les OS propriétaires: *MacOS*, *BeOS*, *NeXTStep*, *MS-DOS* et les différentes variantes de *Windows*. Un nouveau compilateur, baptisé JPython, est écrit en Java et génère du *bytecode* Java.
- Python est **gratuit**, mais on peut l'utiliser sans restriction dans des projets commerciaux.
- Python convient aussi bien à des **scripts** d'une dizaine de lignes qu'à des **projets complexes** de plusieurs dizaines de milliers de lignes.
- La **syntaxe de Python est très simple** et, combinée à des **types de données évolués** (listes, dictionnaires,...), conduit à des programmes à la fois très compacts et très lisibles. A fonctionnalités égales, un programme Python (abondamment commenté et présenté selon les canons standards) est souvent de 3 à 5 fois plus court qu'un programme C ou C++ (ou même Java) équivalent, ce qui représente en général un temps de développement de 5 à 10 fois plus court et une facilité de maintenance largement accrue.
- Python gère ses ressources (mémoire, descripteurs de fichiers...) sans intervention du programmeur, par un mécanisme de **comptage de références** (proche, mais différent, d'un *garbage collector*).
- Il n'y a **pas de pointeurs** explicites en Python.
- Python est (optionnellement) **multi-threadé**.
- Python est **orienté-objet**. Il supporte **l'héritage multiple** et **la surcharge des opérateurs**. Dans son modèle objets, et en reprenant la terminologie de C++, toutes les méthodes sont virtuelles.
- Python intègre, comme Java ou les versions récentes de C++, un système d'**exceptions**, qui permettent de simplifier considérablement la gestion des erreurs.

2 Stéfane Fermigier est le président de l'AFUL (Association Francophone des Utilisateurs de Linux et des logiciels libres). Ce texte est extrait d'un article paru dans le magazine **Programmez!** en décembre 1998. Il est également disponible sur http://www.linux-center.org/articles/9812/python.html)

- Python est **dynamique** (l'interpréteur peut évaluer des chaînes de caractères représentant des expressions ou des instructions Python), **orthogonal** (un petit nombre de concepts suffit à engendrer des constructions très riches), **reflectif** (il supporte la *métaprogrammation*, par exemple la capacité pour un objet de se rajouter ou de s'enlever des attributs ou des méthodes, ou même de changer de classe en cours d'exécution) et **introspectif** (un grand nombre d'outils de développement, comme le *debugger* ou le *profiler*, sont implantés en Python lui-même).
- Comme *Scheme* ou *SmallTalk*, Python est dynamiquement typé. Tout objet manipulable par le programmeur possède un type bien défini à l'exécution, qui n'a pas besoin d'être déclaré à l'avance.
- Python possède actuellement deux implémentations. L'une, **interprétée**, dans laquelle les programmes Python sont compilés en instructions portables, puis exécutés par une machine virtuelle (comme pour Java, avec une différence importante: Java étant statiquement typé, il est beaucoup plus facile d'accélérer l'exécution d'un programme Java que d'un programme Python). L'autre génère directement du *bytecode* Java.
- Python est **extensible** : comme *Tcl* ou *Guile*, on peut facilement l'interfacer avec des bibliothèques C existantes. On peut aussi s'en servir comme d'un langage d'extension pour des systèmes logiciels complexes.
- La **bibliothèque standard** de Python, et les paquetages contribués, donnent accès à une grande variété de services : chaînes de caractères et expressions régulières, services UNIX standard (fichiers, *pipes*, signaux, sockets, threads...), protocoles Internet (Web, News, FTP, CGI, HTML...), persistance et bases de données, interfaces graphiques.
- Python est un langage qui **continue à évoluer**, soutenu par une communauté d'utilisateurs enthousiastes et responsables, dont la plupart sont des supporters du logiciel libre. Parallèlement à l'interpréteur principal, écrit en C et maintenu par le créateur du langage, un deuxième interpréteur, écrit en Java, est en cours de développement.

Distribution de Python - Bibliographie

Les différentes versions de Python (pour Window$, Un!x, etc.), son ***tutoriel*** original, son ***manuel de référence***, la ***documentation*** des bibliothèques de fonctions, etc. sont disponibles en téléchargement gratuit depuis l'internet, à partir du site web officiel : ***http://www.python.org***

Il existe également de très bons ouvrages imprimés concernant Python. Si la plupart d'entre eux n'existent encore qu'en version anglaise, on peut cependant déjà se procurer en traduction française les manuels ci-après :

- ***Introduction à Python***, par Mark Lutz & David Ascher, traduction de Sébastien Tanguy, Olivier Berger & Jérôme Kalifa, Editions O'Reilly, Paris, 2000, 385 p., ISBN 2-84177-089-3
- ***L'intro Python***, par Ivan Van Laningham, traduction de Denis Frère, Karine Cottereaux et Noël Renard, Editions CampusPress, Paris, 2000, 484 p., ISBN 2-7440-0946-6
- ***Python précis & concis*** (il s'agit d'un petit aide-mémoire bien pratique), par Mark Lutz, traduction de James Guérin, Editions O'Reilly, Paris, 2000, 80 p., ISBN 2-84177-111-3

En langue anglaise, le choix est évidemment beaucoup plus vaste. Nous apprécions personnellement beaucoup ***Python : How to program***, par Deitel, Liperi & Wiedermann, Prentice Hall, Upper Saddle River - NJ 07458, 2002, 1300 p., ISBN 0-13-092361-3 , très complet, très clair, agréable à lire et qui utilise une méthodologie éprouvée, ***Core Python programming***, par Wesley J. Chun, Prentice Hall, 2001, 770 p., ISBN 0-13-026036-3 dont les explications sont limpides, et ***Learn to program using Python***, par Alan Gauld, Addison-Wesley, Reading, MA, 2001, 270 p., ISBN 0-201-70938-4 , qui est un très bon ouvrage pour débutants.

Pour aller plus loin, notamment dans l'utilisation de la bibliothèque graphique *Tkinter*, on pourra utilement consulter ***Python and Tkinter Programming***, par John E. Grayson, Manning publications co., Greenwich (USA), 2000, 658 p., ISBN 1-884777-81-3 , et surtout l'incontournable ***Programming Python*** (second edition) de Mark Lutz, Editions O'Reilly, Paris, 2001, 1255 p., ISBN 0-596-00085-5, qui est une extraordinaire mine de renseignements sur de multiples aspects de la programmation moderne (sur tous systèmes).

Si vous souhaitez plus particulièrement exploiter aux mieux les ressources liées au système d'exploitation *Window$*, ***Python Programming on Win32,*** par Mark Hammond & Andy Robinson, Editions O'Reilly, Paris, 2000, 654 p., ISBN 1-56592-621-8 est un ouvrage précieux.

Référence également fort utile, la ***Python Standard Library*** de Fredrik Lundh, Editions O'Reilly, Paris, 2001, 282 p., ISBN 0-596-00096-0

Pour le professeur : organisation du cours

Nous souhaitons avec ces notes ouvrir un maximum de portes.

A notre niveau d'études, il nous paraît important de montrer que la programmation d'un ordinateur est un vaste univers de concepts et de méthodes, dans lequel chacun peut trouver son domaine de prédilection. Nous ne pensons pas que tous nos étudiants doivent apprendre exactement les mêmes choses. Nous voudrions plutôt qu'ils arrivent à développer chacun des compétences quelque peu différentes, qui leur permettent de se valoriser à leurs propres yeux ainsi qu'à ceux de leurs condisciples, et également d'apporter leur contribution spécifique lorsqu'on leur proposera de collaborer à des travaux d'envergure.

De toute manière, notre préoccupation primordiale doit être d'arriver à susciter l'intérêt, ce qui est loin d'être acquis d'avance pour un sujet aussi ardu que la programmation d'un ordinateur. Nous ne voulons pas feindre de croire que nos jeunes élèves vont se passionner d'emblée pour la construction de beaux algorithmes. Nous sommes plutôt convaincus qu'un certain intérêt ne pourra durablement s'installer qu'à partir du moment où ils commenceront à réaliser qu'ils sont devenus capables de développer un projet personnel original, dans une certaine autonomie.

Ce sont ces considérations qui nous ont amenés à développer une structure de cours que certains trouveront peut-être un peu chaotique. Le principal fil conducteur en est l'excellent "*How to think like a computer scientist*", mais nous l'avons un peu éclaté pour y insérer toute une série d'éléments concernant la gestion des entrées/sorties, et en particulier l'interface graphique *Tkinter*. Nous souhaiterions en effet que les élèves puissent déjà réaliser l'une ou l'autre petite application graphique dès la fin de leur première année d'études.

Très concrètement, cela signifie que nous pensons pouvoir explorer les huit premiers chapitres de ces notes durant la première année de cours. Cela suppose que l'on aborde d'abord toute une série de concepts importants (types de données, variables, instructions de contrôle du flux, fonctions et boucles) d'une manière assez rapide, sans trop se préoccuper de ce que chaque concept soit parfaitement compris avant de passer au suivant, en essayant plutôt d'inculquer le goût de la recherche personnelle et de l'expérimentation. Il sera souvent plus efficace de réexpliquer les notions et les mécanismes essentiels en situation, dans des contextes variés.

Dans notre esprit, c'est surtout en seconde année que l'on cherchera à structurer les connaissances acquises, en les approfondissant. Les algorithmes seront davantage décortiqués et commentés. Les projets, cahiers des charges et méthodes d'analyse seront discutés en concertation. On exigera la tenue régulière d'un cahier de notes et la rédaction de rapports techniques pour certains travaux.

L'objectif ultime sera pour chaque élève de réaliser un projet de programmation original d'une certaine importance. On s'efforcera donc de boucler l'étude théorique des concepts essentiels suffisamment tôt dans l'année scolaire, afin que chacun puisse disposer du temps nécessaire.

Il faut bien comprendre que les nombreuses informations fournies dans ces notes concernant une série de domaines particuliers (gestion des interfaces graphiques, des communications, des bases de données, etc.) sont matières facultatives. Ce sont seulement une série de suggestions et de repères que nous avons inclus pour aider les étudiants à choisir et à commencer leur projet personnel de fin d'études. Nous ne cherchons en aucune manière à former des spécialistes d'un certain langage ou d'un certain domaine technique : nous voulons simplement donner un petit aperçu des immenses possibilités qui s'offrent à celui qui se donne la peine d'acquérir une compétence de programmeur.

Remerciements

Ces notes sont pour une partie le résultat d'un travail personnel, mais pour une autre - bien plus importante - la compilation d'informations et d'idées mises à la disposition de tous par des professeurs et des chercheurs bénévoles. Comme déjà signalé plus haut, l'une de mes sources les plus importantes a été le cours de A.Downey, J.Elkner & C.Meyers : *How to think like a computer scientist*. Merci encore à ces professeurs enthousiastes. J'avoue aussi m'être largement inspiré du tutoriel original écrit par Guido van Rossum lui-même (l'auteur principal de Python), ainsi que d'exemples et de documents divers émanant de la (très active) communauté des utilisateurs de Python. Il ne m'est malheureusement pas possible de préciser davantage les références de tous ces textes, mais je voudrais que leurs auteurs soient assurés de toute ma reconnaissance.

Merci également à tous ceux qui oeuvrent au développement de Python, de ses accessoires et de sa documentation, à commencer par Guido van Rossum, bien sûr, mais sans oublier non plus tous les autres (Il sont (mal)heureusement trop nombreux pour que je puisse les citer tous ici).

Merci encore à mes collègues Sabine Gillman, Freddy Klich et Tony Rodrigues, professeurs à l'Institut St. Jean-Berchmans de Liège, qui ont accepté de se lancer dans l'aventure de ce nouveau cours avec leurs élèves, et ont également suggéré de nombreuses améliorations.

Chapitre 1 : Penser comme un programmeur

1.1 L'activité de programmation

Le but de ce cours est de vous apprendre à ***penser et à réfléchir comme un analyste-programmeur***. Ce mode de pensée combine des démarches intellectuelles complexes, similaires à celles qu'accomplissent les mathématiciens, les ingénieurs et les scientifiques.

Comme le mathématicien, l'analyste-programmeur utilise des langages formels pour décrire des raisonnements (ou algorithmes). Comme l'ingénieur, il conçoit des dispositifs, il assemble des composants pour réaliser des mécanismes et il évalue leurs performances. Comme le scientifique, il observe le comportement de systèmes complexes, il ébauche des hypothèses explicatives, il teste des prédictions.

L'activité essentielle d'un analyste-programmeur est la résolution de problèmes.

Il s'agit là d'une compétence de haut niveau, qui implique des capacités et des connaissances diverses : être capable de (re)formuler un problème de plusieurs manières différentes, être capable d'imaginer des solutions innovantes et efficaces, être capable d'exprimer ces solutions de manière claire et complète.

La programmation d'un ordinateur consiste en effet à "expliquer" en détail à une machine ce qu'elle doit faire, en sachant d'emblée qu'elle ne peut pas véritablement "comprendre" un langage humain, mais seulement effectuer un traitement automatique sur des séquences de caractères.
Un programme n'est rien d'autre qu'une suite d'instructions, encodées en respectant de manière très stricte un ensemble de conventions fixées à l'avance que l'on appelle un langage informatique. La machine est ainsi pourvue d'un mécanisme qui décode ces instructions en associant à chaque "mot" du langage une action précise.

Vous allez donc apprendre à programmer, activité déjà intéressante en elle-même parce qu'elle contribue à développer votre intelligence. Mais vous serez aussi amené à utiliser la programmation pour réaliser des projets concrets, ce qui vous procurera certainement de très grandes satisfactions.

1.2 Langage machine, langage de programmation

A strictement parler, un ordinateur n'est rien d'autre qu'une machine effectuant des opérations simples sur des séquences de signaux électriques, lesquels sont conditionnés de manière à ne pouvoir prendre que deux états seulement (par exemple un potentiel électrique maximum ou minimum). Ces séquences de signaux obéissent à une logique du type "tout ou rien" et peuvent donc être considérés conventionnellement comme des suites de nombres ne prenant jamais que les deux valeurs 0 et 1. Un système numérique ainsi limité à deux chiffres est appelé système binaire.

Sachez dès à présent que dans son fonctionnement interne, un ordinateur est totalement incapable de traiter autre chose que des nombres binaires. Toute information d'un autre type doit être convertie, ou codée, ***en format binaire***. Cela est vrai non seulement pour les données que l'on souhaite traiter (les textes, les images, les sons, les nombres, etc.), mais aussi pour programmes, c.à.d. les séquences d'instructions que l'on va fournir à la machine pour lui dire ce qu'elle doit faire avec ces données.

Le seul "langage" que l'ordinateur puisse véritablement "comprendre" est donc très éloigné de ce que nous utilisons nous-mêmes. C'est une longue suite de 1 et de 0 (les "bits") souvent traités par groupes de 8 (les "octets"), 16, 32, ou même 64. Ce "langage machine" est évidemment presqu'incompréhensible pour nous. Pour "parler" à un ordinateur, il nous faudra utiliser des systèmes de traduction automatiques, capables de convertir en nombres binaires des suites de caractères formant des mots-clés (anglais en général) qui seront plus significatifs pour nous.

Ces systèmes de traduction automatique seront établis sur la base de toute une série de conventions, dont il existera évidemment de nombreuses variantes.

Le système de traduction proprement dit s'appellera ***interpréteur*** ou bien ***compilateur***, suivant la méthode utilisée pour effectuer la traduction (voir ci-après). On appellera ***langage de programmation*** un ensemble de mots-clés (choisis arbitrairement) associé à un ensemble de règles très précises indiquant comment on peut assembler ces mots pour former des "phrases" que l'interpréteur ou le compilateur puisse traduire en langage machine (binaire).

Suivant son niveau d'abstraction, on pourra dire d'un langage qu'il est "de bas niveau" (ex : *Assembler*) ou "de haut niveau" (ex : *Pascal, Perl, Smalltalk, Clarion, Java*...). Un langage de bas niveau est constitué d'instructions très élémentaires, très "proches de la machine". Un langage de haut niveau comporte des instructions plus abstraites ou, plus "puissantes". Cela signifie que chacune de ces instructions pourra être traduite par l'interpréteur ou le compilateur en un grand nombre d'instructions machine élémentaires.

Le langage que vous allez apprendre en premier est **Python**. Il s'agit d'un langage de haut niveau, dont la traduction en codes binaires est complexe et prend donc toujours un certain temps. Cela pourrait paraître un inconvénient. En fait, les avantages que présentent les langages de haut niveau sont énormes : il est ***beaucoup plus facile*** d'écrire un programme dans un langage de haut niveau ; l'écriture du programme prend donc beaucoup moins de temps ; la probabilité d'y faire des fautes est nettement plus faible ; la maintenance (c.à.d. l'apport de modifications ultérieures) et la recherche des erreurs (les "bugs") sont grandement facilitées. De plus, un programme écrit dans un langage de haut niveau sera souvent ***portable***, c.à.d. que l'on pourra le faire fonctionner sans guère de modifications sur des machines ou des systèmes d'exploitation différents. Un programme écrit dans un langage de bas niveau ne peut jamais fonctionner que sur un seul type de machine : pour qu'une autre l'accepte, il faut le réécrire entièrement.

1.3 Compilation et interprétation

Le programme tel que nous l'écrivons à l'aide d'un logiciel éditeur (une sorte de traitement de texte spécialisé) sera appelé désormais ***programme source*** (ou code source). Comme déjà signalé plus haut, il existe deux techniques principales pour effectuer la traduction d'un tel programme source en code binaire exécutable par la machine : l'interprétation et la compilation.

- Dans la technique appelée ***interprétation***, le logiciel interpréteur doit être utilisé chaque fois que l'on veut faire fonctionner le programme. Dans cette technique en effet, chaque ligne du programme source analysé est traduite au fur et à mesure en quelques instructions du langage machine, qui sont ensuite directement exécutées. Aucun programme objet n'est généré.



- La ***compilation*** consiste à traduire la totalité du texte source en une fois. Le logiciel compilateur lit toutes les lignes du programme source et produit une nouvelle suite de codes que l'on appelle ***programme objet*** (ou code objet). Celui-ci peut désormais être exécuté indépendamment du compilateur et être conservé tel quel dans un fichier ("fichier exécutable").



Chacune de ces deux techniques a ses avantages et ses inconvénients :

L'interprétation est idéale lorsque l'on est en phase d'apprentissage du langage, ou en cours d'expérimentation sur un projet. Avec cette technique, on peut en effet tester immédiatement toute modification apportée au programme source, sans passer par une phase de compilation qui demande toujours du temps.

Par contre, lorsqu'un projet comporte des fonctionnalités complexes qui doivent s'exécuter rapidement, la compilation est préférable : il est clair en effet qu'un programme compilé fonctionnera toujours nettement plus vite que son homologue interprété, puisque dans cette technique l'ordinateur n'a plus à (re)traduire chaque instruction en code binaire avant qu'elle puisse être exécutée.

Certains langages modernes tentent de combiner les deux techniques afin de retirer le meilleur de chacune. C'est le cas de Python et aussi de Java. Lorsque vous lui fournissez un programme source, Python commence par le compiler pour produire un code intermédiaire, similaire à un langage machine, que l'on appelle ***bytecode***, lequel sera ensuite transmis à un interpréteur pour l'exécution finale. Du point de vue de l'ordinateur, le *bytecode* est très facile à interpréter en langage machine. Cette interprétation sera donc beaucoup plus rapide que celle d'un code source.



Les avantages de cette méthode sont appréciables :

- Le fait de disposer en permanence d'un interpréteur permet de tester immédiatement n'importe quel petit morceau de programme. On pourra donc vérifier le bon fonctionnement de chaque composant d'une application au fur et à mesure de sa construction.
- L'interprétation du *bytecode* compilé n'est pas aussi rapide que celle d'un véritable code binaire, mais elle est très satisfaisante pour de très nombreux programmes, y compris graphiques.
- Le *bytecode* est ***portable***. Pour qu'un programme Python ou Java puisse s'exécuter sur différentes machines, il suffit de disposer pour chacune d'elles d'un interpréteur adapté.

Tout ceci peut vous paraître un peu compliqué, mais la bonne nouvelle est que tout ceci est pris en charge automatiquement par l'environnement de développement de Python. Il vous suffira d'entrer vos commandes au clavier, de frapper <Enter>, et Python se chargera de les compiler et de les interpréter pour vous.

1.4 Mise au point d'un programme - Recherche des erreurs ("debug")

La programmation est une démarche très complexe, et comme c'est le cas dans toute activité humaine, on y commet de nombreuses erreurs. Pour des raisons anecdotiques, les erreurs de programmation s'appellent des "*bugs*" (ou "bogues", en France)[3], et l'ensemble des techniques que l'on met en oeuvre pour les détecter et les corriger s'appelle "*debug*" (ou "déboguage").

En fait, il peut exister dans un programme trois types d'erreurs assez différentes, et il convient que vous appreniez à bien les distinguer :

1.4.1 Erreurs de syntaxe

Python ne peut exécuter un programme que si sa ***syntaxe*** est parfaitement correcte. Dans le cas contraire, le processus s'arrête et vous obtenez un message d'erreur. Le terme syntaxe se réfère aux règles que les auteurs du langage ont établies pour la structure du programme.

Tout langage comporte sa syntaxe. Dans la langue française, par exemple, une phrase doit toujours commencer par une majuscule et se terminer par un point. ainsi cette phrase comporte deux erreurs de syntaxe

Dans les textes ordinaires, la présence de quelques petites fautes de syntaxe par-ci par-là n'a généralement pas d'importance. Il peut même arriver (en poésie, par exemple), que des fautes de syntaxe soient commises volontairement. Cela n'empêche pas que l'on puisse comprendre le texte.

Dans un programme d'ordinateur, par contre, la moindre erreur de syntaxe produit invariablement un arrêt de fonctionnement (un "plantage") ainsi que l'affichage d'un message d'erreur. Au cours des premières semaines de votre carrière de programmeur, vous passerez certainement pas mal de temps à rechercher vos erreurs de syntaxe. Avec de l'expérience, vous en commettrez beaucoup moins.

Gardez à l'esprit que les mots et les symboles utilisés n'ont aucune signification en eux-mêmes : ce ne sont que des suites de codes destinés à être convertis automatiquement en nombres binaires. Par conséquent, il vous faudra être très attentifs à respecter scrupuleusement la syntaxe du langage.

Il est heureux que vous fassiez vos débuts en programmation avec un langage interprété tel que Python. La recherche des erreurs y est facile et rapide. Avec les langages compilés (tel C++), il vous faudrait recompiler l'entièreté du programme après chaque modification, aussi minime soit-elle.

1.4.2 Erreurs sémantiques

Le second type d'erreur est l'erreur ***sémantique*** ou erreur de logique. S'il existe une erreur de ce type dans un de vos programmes, celui-ci s'exécute parfaitement, en ce sens que vous n'obtenez aucun message d'erreur, mais le résultat n'est pas celui que vous attendiez : vous obtenez autre chose.

En réalité, le programme fait exactement ce que vous lui avez dit de faire. Le problème est que ce que vous lui avez dit de faire ne correspond pas à ce que vous vouliez qu'il fasse. La séquence d'instructions de votre programme ne correspond pas à l'objectif poursuivi. La sémantique (la logique) est incorrecte.

Rechercher des fautes de logique peut être une tâche ardue. Il faut analyser ce qui sort de la machine et tâcher de se représenter une par une les opérations qu'elle a effectuées, à la suite de chaque instruction.

3 "*bug*" est à l'origine un terme anglais servant à désigner de petits insectes gênants, tels les punaises. Les premiers ordinateurs fonctionnaient à l'aide de "lampes" radios qui nécessitaient des tensions électriques assez élevées. Il est arrivé à plusieurs reprises que des petits insectes s'introduisent dans cette circuiterie complexe et se fassent électrocuter, leurs cadavres calcinés provoquant alors des court-circuits et donc des pannes incompréhensibles.
Le mot français "*bogue*" a été choisi par homonymie approximative. Il désigne la coque épineuse de la châtaigne.

1.4.3 Erreurs à l'exécution

Le troisième type d'erreur est l'erreur en cours d'exécution (*Run-time error*), qui apparaît seulement lorsque votre programme fonctionne déjà, mais que des circonstances particulières se présentent (par exemple, votre programme essaie de lire un fichier qui n'existe plus). Ces erreurs sont également appelées des ***exceptions***, parce qu'elles indiquent généralement que quelque chose d'exceptionnel s'est produit (et qui n'avait pas été prévu). Vous rencontrerez davantage ce type d'erreur lorsque vous programmerez des projets de plus en plus volumineux.

1.5 Recherche des erreurs et expérimentation

L'une des compétences les plus importantes à acquérir au cours de votre apprentissage est celle qui consiste à "déboguer" efficacement un programme. Il s'agit d'une activité intellectuelle parfois énervante mais toujours très riche, dans laquelle il faut faire montre de beaucoup de perspicacité.

Ce travail ressemble par bien des aspects à une enquête policière. Vous examinez un ensemble de faits, et vous devez émettre des hypothèses explicatives pour reconstituer les processus et les événements qui ont logiquement entraîné les résultats que vous constatez.

Cette activité s'apparente aussi au travail expérimental en sciences. Vous vous faites une première idée de ce qui ne va pas, vous modifiez votre programme et vous essayez à nouveau. Vous avez émis une hypothèse, qui vous permet de prédire ce que devra donner la modification. Si la prédiction se vérifie, alors vous avez progressé d'un pas sur la voie d'un programme qui fonctionne. Si la prédiction se révèle fausse, alors il vous faut émettre une nouvelle hypothèse. Comme l'a bien dit Sherlock Holmes : "*Lorsque vous avez éliminé l'impossible, ce qui reste, même si c'est improbable, doit être la vérité*" (A. Conan Doyle, *Le signe des quatre*).

Pour certaines personnes, "programmer" et "déboguer" signifient exactement la même chose.
Ce qu'elles veulent dire par là est que l'activité de programmation consiste en fait à modifier, à corriger sans cesse un même programme, jusqu'à ce qu'il se comporte finalement comme vous le vouliez. L'idée est que la construction d'un programme commence toujours par une ébauche qui fait déjà quelque chose (et qui est donc déjà déboguée), à laquelle on ajoute couche par couche de petites modifications, en corrigeant au fur et à mesure les erreurs, afin d'avoir de toute façon à chaque étape du processus un programme qui fonctionne.

Par exemple, vous savez que Linux est un système d'exploitation (et donc un gros logiciel) qui comporte des milliers de lignes de code. Au départ, cependant, cela a commencé par un petit programme simple que Linus Torvalds avait développé pour tester les particularités du processeur Intel 80386. Suivant Larry GreenField ("*The Linux user's guide*", beta version 1) : "*L'un des premiers projets de Linus était un programme destiné à convertir une chaîne de caractères AAAA en BBBB. C'est cela qui plus tard finit par devenir Linux !*".

Ce qui précède ne signifie pas que nous voulions vous pousser à programmer par approximations successives, à partir d'une vague idée. Lorsque vous démarrerez un projet de programmation d'une certaine importance, il faudra au contraire vous efforcer d'établir le mieux possible ***un cahier des charges détaillé***, lequel s'appuiera sur un plan solidement construit pour l'application envisagée.

Diverses méthodes existent pour effectuer cette tâche d'***analyse***, mais leur étude sort du cadre de ces notes. Veuillez consulter votre professeur pour de plus amples informations et références.

1.6 Langages naturels et langages formels

Les ***langages naturels*** sont ceux que les êtres humains utilisent pour communiquer. Ces langages n'ont pas été mis au point délibérément (encore que certaines instances tâchent d'y mettre un peu d'ordre) : ils évoluent naturellement.

Les ***langages formels*** sont des langages développés en vue d'applications spécifiques. Ainsi par exemple, le système de notation utilisé par les mathématiciens est un langage formel particulièrement efficace pour représenter les relations entre nombres et grandeurs diverses. Les chimistes utilisent un langage formel pour représenter la structure des molécules, etc.

> ***Les langages de programmation sont des langages formels qui ont été développés pour décrire des algorithmes.***

Comme on l'a déjà signalé plus haut, les langages formels sont dotés d'une syntaxe qui obéit à des règles très strictes. Par exemple, `3+3=6` est une représentation mathématique correcte, alors que `$3=+6` ne l'est pas. De même, la formule chimique $\mathbf{H_2O}$ est correcte, mais non $\mathbf{Zq_3G_2}$

Les règles de syntaxe s'appliquent non seulement aux symboles du langage (par exemple, le symbole chimique Zq est illégal parce qu'il ne correspond à aucun élément), mais aussi à la manière de les combiner. Ainsi l'équation mathématique `6+=+/5-` ne contient que des symboles parfaitement autorisés, mais leur arrangement incorrect ne signifie rien du tout.

Lorsque vous lisez une phrase quelconque, vous devez arriver à vous représenter la structure logique de la phrase (même si vous faites cela inconsciemment la plupart du temps). Par exemple, lorsque vous lisez la phrase "*la pièce est tombée*", vous comprenez que "*la pièce*" en est le sujet et "*est tombée*" le verbe. L'analyse vous permet de comprendre la signification, la logique de la phrase (sa sémantique). D'une manière analogue, l'interpréteur Python devra ***analyser*** la structure de votre programme source pour en extraire la signification.

Les langages naturels et formels ont donc beaucoup de caractéristiques communes (des symboles, une syntaxe, une sémantique), mais ils présentent aussi des différences très importantes :

Ambiguïté.

Les langages naturels sont pleins d'ambiguïtés, que nous pouvons lever dans la plupart des cas en nous aidant du contexte. Par exemple, nous attribuons tout naturellement une signification différente au mot vaisseau, suivant que nous le trouvons dans une phrase qui traite de circulation sanguine ou de navigation à voiles. Dans un langage formel, il ne peut pas y avoir d'ambiguïté. Chaque instruction possède une seule signification, indépendante du contexte.

Redondance.

Pour compenser toutes ces ambiguïtés et aussi de nombreuses erreurs ou pertes dans la transmission de l'information, les langages naturels emploient beaucoup la redondance (dans nos phrases, nous répétons plusieurs fois la même chose sous des formes différentes, pour être sûrs de bien nous faire comprendre). Les langages formels sont beaucoup plus concis.

Littéralité.

Les langages naturels sont truffés d'images et de métaphores. Si je dis "*la pièce est tombée !*" dans un certain contexte, il se peut qu'il ne s'agisse en fait ni d'une véritable pièce, ni de la chute de quoi que ce soit. Dans un langage formel, par contre, les expressions doivent être prises pour ce qu'elles sont, "au pied de la lettre".

Habitués comme nous le sommes à utiliser des langages naturels, nous avons souvent bien du mal à nous adapter aux règles rigoureuses des langages formels. C'est l'une des difficultés que vous devrez surmonter pour arriver à penser comme un analyste-programmeur efficace.

Pour bien nous faire comprendre, comparons encore différents types de textes :

Un texte poétique :

Les mots y sont utilisés autant pour leur musicalité que pour leur signification, et l'effet recherché est surtout émotionnel. Les métaphores et les ambiguïtés y règnent en maîtres.

Un texte en prose :

La signification littérale des mots y est plus importante, et les phrases sont structurées de manière à lever les ambiguïtés, mais sans y parvenir toujours complètement. Les redondances sont souvent nécessaires.

Un programme d'ordinateur :

La signification du texte est unique et littérale. Elle peut être comprise entièrement par la seule analyse des symboles et de la structure. On peut donc automatiser cette analyse.

Pour conclure, voici quelques suggestions concernant la manière de lire un programme d'ordinateur (ou tout autre texte écrit en langage formel).

Premièrement, gardez à l'esprit que les langages formels sont beaucoup plus denses que les langages naturels, ce qui signifie qu'il faut davantage de temps pour les lire. De plus, la structure y est très importante. Aussi, ce n'est généralement pas une bonne idée que d'essayer de lire un programme d'une traite, du début à la fin. Au lieu de cela, entraînez-vous à analyser le programme dans votre tête, en identifiant les symboles et en interprétant la structure.

Finalement, souvenez-vous que tous les détails ont de l'importance. Il faudra en particulier faire très attention à la ***casse*** (c.à.d. l'emploi des majuscules et des minuscules) et à la ***ponctuation***. Toute erreur à ce niveau (même minime en apparence, tel l'oubli d'une virgule, par exemple) peut modifier considérablement la signification du code, et donc le déroulement du programme.

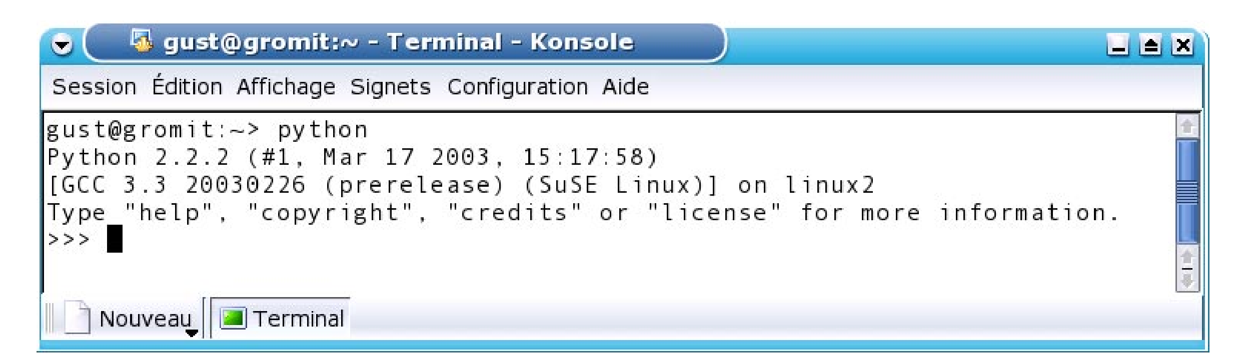
Chapitre 2 : Premières instructions

2.1 Calculer avec Python

Python présente la particularité de pouvoir être utilisé de plusieurs manières différentes. Vous allez d'abord l'utiliser ***en mode interactif***, c.à.d. d'une manière telle que vous pourrez dialoguer avec lui directement depuis le clavier. Cela vous permettra de découvrir très vite un grand nombre de fonctionnalités du langage. Dans un second temps, vous apprendrez comment créer vos premiers programmes (scripts) et les sauvegarder sur disque.

L'interpréteur peut être lancé directement depuis la ligne de commande (dans un "shell" *Linux*, ou bien dans une fenêtre *DOS* sous *Window$*) : il suffit d'y taper la commande **"python"** (en supposant que le logiciel lui-même ait été correctement installé).

Si vous utilisez une interface graphique telle que *Window$*, *Gnome, WindowMaker* ou *KDE*, vous préférerez vraisemblablement travailler dans une "fenêtre de terminal", ou encore dans un environnement de travail spécialisé tel que ***IDLE***. Voici par exemple ce qui apparaît dans une fenêtre de terminal KDE (sous Linux)[4] :



Avec *IDLE* sous *Window$*, votre environnement de travail ressemblera à celui-ci :

```
*Python Shell*
File Edit Debug Windows Help
Python 2.2 (#28, Dec 21 2001, 12:21:22) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
IDLE 0.8 -- press F1 for help
>>>
```

Les trois caractères "supérieur à" constituent le signal d'invite, ou *prompt principal*, lequel vous indique que Python est prêt à exécuter une commande.

4 Sous *Window$,* vous aurez surtout le choix entre l'environnement *IDLE* développé par Guido Van Rossum, auquel nous donnons nous-même la préférence, et *PythonWin*, une interface de développement développée par Mark Hammond. D'autres environnements de travail plus sophistiqués existent aussi, tels l'excellent *Boa Constructor* par exemple (qui fonctionne de façon très similaire à *Delphi*), mais nous estimons qu'ils ne conviennent guère aux débutants. Pour tout renseignement complémentaire, veuillez consulter le site Web de Python.

Sous *Linux*, nous préférons personnellement travailler dans l'environnement graphique *WindowMaker* (plutôt que *KDE* ou *Gnome* trop gourmands en ressources), en ouvrant une simple fenêtre de terminal pour lancer l'interpréteur Python ou l'exécution des scripts, et en faisant appel à l'excellent logiciel *Nedit* pour l'édition de ces derniers.

Par exemple, vous pouvez tout de suite utiliser l'interpréteur comme une simple calculatrice de bureau. Veuillez donc vous-même tester les commandes ci-dessous (Prenez l'habitude d'utiliser votre cahier d'exercices pour noter les résultats qui apparaissent à l'écran) :

```
>>> 5+3

>>> 2 - 9                 # les espaces sont optionnels

>>> 7 + 3 * 4             # la hiérarchie des opérations mathématiques
                          # est-elle respectée ?
>>> (7+3)*4

>>> 20 / 3                # surprise !!!
```

Comme vous pouvez le constater, les opérateurs arithmétiques pour l'addition, la soustraction, la multiplication et la division sont respectivement +, -, * et /. Les parenthèses sont fonctionnelles.

Par défaut, la division est cependant une *division entière*, ce qui signifie que si on lui fournit des arguments qui sont des nombres entiers, le résultat de la division est lui-même un entier (tronqué), comme dans le dernier exemple ci-dessus. Si vous voulez qu'un argument soit compris par Python comme étant un nombre réel, il faut le lui faire savoir, en fournissant au moins un point décimal[5].

Essayez par exemple :

```
>>> 20.0 / 3              # (comparez le résultat avec celui obtenu à l'exercice précédent)

>>> 8./5
```

Si une opération est effectuée avec des arguments de types mélangés (entiers et réels), Python convertit automatiquement les opérandes en réels avant d'effectuer l'opération. Essayez :

```
>>> 4 * 2.5 / 3.3
```

5 Dans tous les langages de programmation, les conventions mathématiques de base sont celles en vigueur dans les pays anglophones : le séparateur décimal sera donc toujours un point, et non une virgule comme chez nous.
Dans le monde de l'informatique, les nombres réels sont souvent désignés comme des nombres "à virgule flottante", ou encore des nombres "de type ***float***".

2.2 Données et variables

Nous aurons l'occasion de détailler plus loin les différents types de données numériques. Mais avant cela, nous pouvons dès à présent aborder un concept de grande importance :

L'essentiel du travail effectué par un programme d'ordinateur consiste à manipuler des ***données***. Ces données peuvent être très diverses (tout ce qui est *numérisable*, en fait[6]), mais dans la mémoire de l'ordinateur elles se ramènent toujours en définitive à ***une suite finie de nombres binaires***.

Pour pouvoir accéder aux données, le programme d'ordinateur (quel que soit le langage dans lequel il est écrit) fait abondamment usage d'un grand nombre de ***variables*** de différents types.

Une variable apparaît dans un langage de programmation sous un ***nom de variable*** à peu près quelconque (voir ci-après), mais pour l'ordinateur il s'agit d'une ***référence*** désignant une ***adresse mémoire***, c.à.d. un emplacement précis dans la mémoire vive.

A cet emplacement est stocké une ***valeur*** bien déterminée. C'est la donnée proprement dite, qui est donc stockée sous la forme d'une suite de nombres binaires, mais qui n'est pas nécessairement un nombre aux yeux du langage de programmation utilisé. Cela peut être en fait à peu près n'importe quel "objet" susceptible d'être placé dans la mémoire d'un ordinateur, comme par exemple : un nombre entier, un nombre réel, un nombre complexe, un vecteur, une chaîne de caractères typographiques, un tableau, une fonction, etc.

Pour distinguer les uns des autres ces divers contenus possibles, le langage de programmation fait usage de différents ***types de variables***. *(le type 'entier', le type 'réel', le type 'chaîne de caractères', le type 'liste', etc.)*. Nous allons expliquer tout cela dans les pages suivantes.

6 *Que peut-on numériser au juste ?* Voilà une question très importante, qu'il vous faudra débattre dans votre cours d'informatique générale.

2.3 Noms de variables et mots réservés

Les noms de variables sont des noms que vous choisissez vous-même assez librement. Efforcez-vous cependant de bien les choisir : de préférence assez courts, mais aussi explicites que possible, de manière à exprimer clairement ce que la variable est censée contenir. Par exemple, des noms de variables tel que ***altitude***, ***altit*** ou ***alt*** conviennent mieux que ***x*** pour exprimer une altitude.

Un bon programmeur doit veiller à ce que ses lignes d'instructions soient faciles à lire.

Sous Python, les noms de variables doivent en outre obéir à quelques règles simples :

- Un nom de variable est une séquence de lettres (a → z , A → Z) et de chiffres (0 → 9), qui doit toujours commencer par une lettre.
- Seules les lettres ordinaires sont autorisées. Les lettres accentuées, les cédilles, les espaces, les caractères spéciaux tels que $, #, @, etc. sont interdits, à l'exception du caractère _ (souligné).
- La casse est significative (c.à.d. que les caractères majuscules et minuscules sont distingués). ***Attention :*** *Joseph, joseph, JOSEPH sont donc des variables différentes. Soyez attentifs !*

Prenez l'habitude d'écrire l'essentiel des noms de variables en caractères minuscules (y compris la première lettre[7]). Il s'agit d'une simple convention, mais elle est largement respectée. N'utilisez les majuscules qu'à l'intérieur même du nom, pour en augmenter éventuellement la lisibilité, comme dans *tableDesMatieres*, par exemple.

En plus de ces règles, il faut encore ajouter que vous ne pouvez pas utiliser comme noms de variables les 28 "mots réservés" ci-dessous (ils sont utilisés par le langage lui-même) :

and	**continue**	**else**	**for**	**import**	**not**	**raise**
assert	**def**	**except**	**from**	**in**	**or**	**return**
break	**del**	**exec**	**global**	**is**	**pass**	**try**
class	**elif**	**finally**	**if**	**lambda**	**print**	**while**

7 Les noms commençant par une majuscule ne sont pas interdits, mais l'usage veut qu'on le réserve plutôt aux variables qui désignent des ***classes*** (le concept de classe sera abodé plus loin dans ces notes).

2.4 Affectation (ou assignation)

Nous savons désormais comment choisir judicieusement un nom de variable. Voyons à présent comment nous pouvons en définir une et lui ***affecter*** une valeur. Les termes "affecter une valeur" ou "assigner une valeur" à une variable sont équivalents. Ils désignent l'opération par laquelle on établit un lien entre le nom de la variable et sa valeur (son contenu).

En Python comme dans de nombreux autres langages, l'opération d'affectation est représentée par le signe ***égale***[8] :

```
>>> n = 7                        # donner à n la valeur 7
>>> msg = "Quoi de neuf ?"       # affecter la valeur "Quoi de neuf ?" à msg
>>> pi = 3.14159                 # assigner sa valeur à la variable pi
```

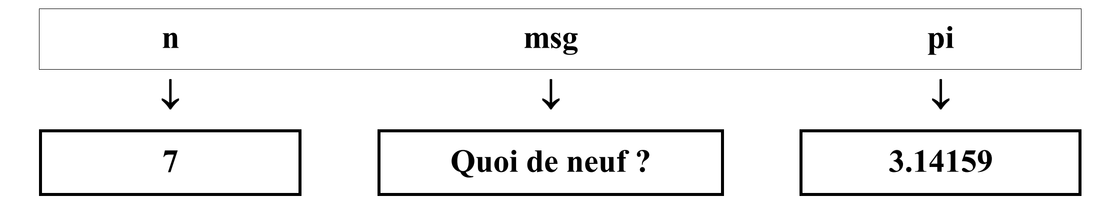
Les exemples ci-dessus illustrent des instructions d'affectation Python tout à fait classiques. Après qu'on les ait exécutées, il existe dans la mémoire de l'ordinateur, à des endroits différents :

- trois noms de variables, à savoir **n**, **msg** et **pi**
- trois séquences d'octets, où sont encodées le nombre entier **7**, la chaîne de caractères **Quoi de neuf ?** et le nombre réel **3,14159**.

Les trois instructions d'affectation ci-dessus ont eu pour effet chacune de réaliser plusieurs opérations dans la mémoire de l'ordinateur :

- créer et mémoriser un nom de variable ;
- lui attribuer un type bien déterminé (ce point sera explicité à la page suivante) ;
- créer et mémoriser une valeur particulière ;
- établir un lien (par un système interne de pointeurs) entre le nom de la variable et l'emplacement mémoire de la valeur correspondante.

On peut mieux se représenter tout cela par un **diagramme d'état** tel que celui-ci :



Les trois noms de variables sont des ***références***, mémorisées dans une zone particulière de la mémoire que l'on appelle ***espace de noms***, alors que les valeurs correspondantes sont situées ailleurs. Nous aurons l'occasion de préciser ce concept plus loin dans ces pages.

8 Il faut bien comprendre qu'il ne s'agit en aucune façon d'une égalité, et que l'on aurait très bien pu choisir un autre symbolisme, tel que **n** ← **7** par exemple, comme on le fait souvent dans certains pseudo-langages servant à décrire des algorithmes, pour bien montrer qu'il s'agit de relier un contenu (la valeur 7) à un contenant (la variable n).

2.5 Afficher la valeur d'une variable

A la suite de l'exercice ci-dessus, nous disposons donc des trois variables **n**, **msg** et **pi**.
Pour afficher leur valeur à l'écran, il existe deux possibilités. La première consiste à entrer au clavier le nom de la variable, puis <Enter>. Python répond en affichant la valeur correspondante :

```
>>> n
7
>>> msg
"Quoi de neuf ?"
>>> pi
3.14159
```

Il s'agit cependant là d'une fonctionnalité secondaire de l'interpréteur, qui est destinée à vous faciliter la vie lorsque vous faites de simples exercices à la ligne de commande. A l'intérieur d'un programme, vous utiliserez toujours l'instruction **print** :

```
>>> print msg
Quoi de neuf ?
```

Remarquez la subtile différence dans les affichages obtenus avec chacune des deux méthodes. L'instruction **print** n'affiche strictement que la valeur de la variable, telle qu'elle a été encodée, alors que l'autre méthode (celle qui consiste à entrer seulement le nom de la variable) affiche aussi des guillemets (afin de vous rappeler le type de la variable : nous y reviendrons).

2.6 Typage des variables

Sous Python, il n'est pas nécessaire d'écrire des lignes de programme spécifiques pour définir le type des variables avant de pouvoir les utiliser. Il vous suffit en effet d'assigner une valeur à un nom de variable pour que celle-ci soit ***automatiquement créée avec le type qui correspond au mieux à la valeur fournie***. Dans l'exercice précédent, par exemple, les variables **n**, **msg** et **pi** ont été créées automatiquement chacune avec un type différent ("nombre entier" pour **n**, "chaîne de caractères" pour **msg**, "nombre à virgule flottante" (ou "*float*", en anglais) pour **pi**).

Ceci constitue une particularité intéressante de Python, qui le rattache à une famille particulière de langages où l'on trouve aussi par exemple *Lisp*, *Scheme*, et quelques autres. On dira à ce sujet que ***le typage des variables sous Python est un typage dynamique***, par opposition au ***typage statique*** qui est de règle par exemple en *C++* ou en *Java*. Dans ces langages, il faut toujours - par des instructions distinctes - d'abord déclarer (définir) le nom et le type des variables, et ensuite seulement leur assigner un contenu, lequel doit bien entendu être compatible avec le type déclaré.

Le typage statique est préférable dans le cas des langages compilés, parce qu'il permet d'optimiser l'opération de compilation (dont le résultat est un code binaire "figé").

Le typage dynamique quant à lui permet d'écrire plus aisément des constructions logiques de niveau élevé (métaprogrammation, réflexivité), en particulier dans le contexte de la programmation orientée objet (polymorphisme). Il facilite également l'utilisation de structures de données très riches telles que les listes et les dictionnaires.

2.7 Affectations multiples

Sous Python, on peut assigner une valeur à plusieurs variables simultanément. Exemple :

```
>>> x = y = 7
>>> x
7
>>> y
7
```

On peut aussi effectuer des ***affectations parallèles*** à l'aide d'un seul opérateur :

```
>>> a, b = 4, 8.33
>>> a
4
>>> b
8.33
```

Dans cet exemple, les variables **a** et **b** prennent simultanément les nouvelles valeurs 4 et 8,33.

Attention *: les francophones que nous sommes avons pour habitude d'utiliser la virgule comme séparateur décimal, alors que les langages de programmation utilisent toujours la convention en vigueur dans les pays de langue anglaise, c.à.d. le* ***point*** *décimal. La* ***virgule****, quant à elle, est très généralement utilisée pour séparer différents éléments (arguments, etc.) comme on le voit dans notre exemple, pour les variables elles-mêmes ainsi que pour les valeurs qu'on leur attribue.*

Exercices :

e 1. Décrivez le plus clairement et le plus complètement possible ce qui se passe à chacune des trois lignes de l'exemple ci-dessous :
>>> **largeur = 20**
>>> **hauteur = 5 * 9.3**
>>> **largeur * hauteur**
930

e 2. Assignez les valeurs respectives 3, 5, 7 à trois variables a, b, c.
Effectuez l'opération a - b/c . Le résultat est-il mathématiquement correct ?
Si ce n'est pas le cas, comment devez-vous procéder pour qu'il le soit ?

2.8 Opérateurs et expressions

On manipule les valeurs et les variables qui les référencent, en les combinant avec des ***opérateurs*** pour former des ***expressions***. Exemple :

```
a, b = 7.3, 12
y = 3*a + b/5
```

Dans cet exemple, nous commençons par affecter aux variables **a** et **b** les valeurs **7,3** et **12.** Comme déjà expliqué précédemment, Python assigne automatiquement le type "réel" à la variable **a**, et le type "entier" à la variable **b**.

La seconde ligne de l'exemple consiste à affecter à une nouvelle variable **y** le résultat d'une expression qui combine les ***opérateurs*** ***** , **+** et **/** avec les ***opérandes*** **a, b, 3** et **5**. Les opérateurs sont les symboles spéciaux utilisés pour représenter des opérations mathématiques simples, telles l'addition ou la multiplication. Les opérandes sont les valeurs combinées à l'aide des opérateurs.

Python évalue chaque expression qu'on lui soumet, aussi compliquée soit-elle, et le résultat de cette évaluation est toujours lui-même une valeur. A cette valeur, il attribue automatiquement un type, lequel dépend de ce qu'il y a dans l'expression. Dans l'exemple ci-dessus, **y** sera du type "réel", parce que l'expression évaluée pour déterminer sa valeur contient elle-même au moins un réel.

Les opérateurs Python ne sont pas seulement les quatre opérateurs mathématiques de base. Il faut leur ajouter l'opérateur ** pour l'exponentiation, ainsi qu'un certain nombre d'opérateurs logiques, des opérateurs agissant sur les chaînes de caractères, des opérateurs effectuant des tests d'identité ou d'appartenance, etc. Nous reparlerons de tout cela plus loin.

Signalons au passage la disponibilité de l'opérateur ***modulo***, représenté par le symbole **%**.
Cet opérateur fournit ***le reste de la division entière*** d'un nombre par un autre. Essayez par exemple :

```
>>> 10 % 3             (et prenez note de ce qui se passe !)
>>> 10 % 5
```

Cet opérateur vous sera très utile plus loin, notamment pour tester si un nombre **a** est divisible par un nombre **b**. Il suffira en effet de vérifier que **a % b** donne un résultat égal à zéro.

Exercice :

e 3. Testez les lignes d'instructions suivantes. Décrivez dans votre cahier ce qui se passe :

```
>>>  r , pi = 12, 3.14159
>>>  s = pi * r**2
>>>  print s
>>>  print type(r), type(pi), type(s)
>>>
```

Quelle est, à votre avis, l'utilité de la *fonction* **type()** ?
(Note : les *fonctions* seront décrites en détail, plus loin dans ce cours).

2.9 Priorité des opérations

Lorsqu'il y a plus d'un opérateur dans une expression, l'ordre dans lequel les opérations doivent être effectuées dépend de ***règles de priorité***. Sous Python, les règles de priorité sont les mêmes que celles qui vous ont été enseignées au cours de mathématique. Vous pouvez les mémoriser aisément à l'aide d'un "truc" mnémotechnique, l'acronyme **PEMDAS** :

- **P** pour ***parenthèses***. Ce sont elles qui ont la plus haute priorité. Elles vous permettent donc de "forcer" l'évaluation d'une expression dans l'ordre que vous voulez.
  Ainsi `2*(3-1) = 4` , et `(1+1)**(5-2) = 8`.
- **E** pour ***exposants***. Les exposants sont évalués ensuite, avant les autres opérations.
  Ainsi `2**1+1 = 3` (et non 4), et `3*1**10 = 3` (et non 59049 !).
- **M** et **D** pour ***multiplication*** et ***division***, qui ont la même priorité. Elles sont évaluées avant l'***addition*** **A** et la ***soustraction*** **S**, lesquelles sont donc effectuées en dernier lieu.
  Ainsi `2*3-1 = 5` (plutôt que 4), et `2/3-1 = -1` (Rappelez-vous que par défaut Python effectue une division ***entière***).
- Si deux opérateurs ont la même priorité, l'évaluation est effectuée de gauche à droite.
  Ainsi dans l'expression `59*100/60`, la multiplication est effectuée en premier, et la machine doit donc ensuite effectuer `5900/60`, ce qui donne `98`. Si la division était effectuée en premier, le résultat serait `59` (rappelez-vous ici encore qu'il s'agit d'une division entière).

2.10 Composition

Jusqu'ici nous avons examiné les différents éléments d'un langage de programmation, à savoir : les ***variables***, les ***expressions*** et les ***instructions***, mais sans traiter de la manière dont nous pouvons les combiner les unes avec les autres.

Or l'une des grandes forces d'un langage de programmation de haut niveau est qu'il permet de construire des instructions complexes par assemblage de fragments divers. Ainsi par exemple, si vous savez comment additionner deux nombres et comment afficher une valeur, vous pouvez combiner ces deux instructions en une seule :

```
>>> print 17 + 3
>>> 20
```

Cela n'a l'air de rien, mais cette fonctionnalité qui paraît si évidente va vous permettre de programmer des algorithmes complexes de façon claire et concise. Exemple :

```
>>> h, m, s = 15, 27, 34
>>> print "nombre de secondes écoulées depuis minuit = ", h*3600 + m*60 + s
```

Attention cependant : il y a une limite à ce que vous pouvez combiner ainsi :

Ce que vous placez à la gauche du signe égale dans une expression doit toujours être une variable, et non une expression. Cela provient du fait que le signe égale n'a pas ici la même signification qu'en mathématique : comme nous l'avons déjà signalé, il s'agit d'un symbole d'affectation (nous plaçons un certain contenu dans une variable) et non un symbole d'égalité. Le symbole d'égalité (dans un test conditionnel, par exemple) sera évoqué un peu plus loin.

Ainsi par exemple, l'instruction **m + 1 = b** est tout à fait ***illégale***.

Par contre, écrire **a = a + 1** est inacceptable en mathématique, alors que cette forme d'écriture est très fréquente en programmation. L'instruction **a = a + 1** signifie en l'occurrence "augmenter la valeur de la variable **a** d'une unité" (ou encore : "incrémenter **a**").

Nous aurons l'occasion de revenir bientôt sur ce sujet. Mais auparavant, il nous faut encore aborder un autre concept de grande importance.

Chapitre 3 : Contrôle du flux d'instructions

Un programme d'ordinateur comporte généralement un grand nombre d'instructions. Mais dans quel ordre ces instructions doivent-elles être exécutées ?

3.1 Suite (ou séquence[9]) d'instructions

Sauf mention explicite, les instructions d'un programme s'exécutent les unes après les autres, dans l'ordre où elles ont été écrites à l'intérieur du script.

Le "chemin" suivi par Python à travers un programme est appelé un ***flux d'instructions***, et les constructions qui le modifient sont appelées des ***instructions de contrôle de flux***.

Python exécute normalement les instructions de la première à la dernière, sauf lorsqu'il rencontre une ***instruction conditionnelle*** comme l'instruction **if** décrite ci-après (nous en rencontrerons d'autres plus loin, notamment à propos des boucles). Une telle instruction va permettre au programme de suivre différents chemins suivant les circonstances.

3.2 Exécution conditionnelle

Si nous voulons pouvoir écrire des applications véritablement utiles, il nous faut des techniques permettant d'aiguiller le déroulement du programme dans différentes directions, en fonction des circonstances rencontrées. Pour ce faire, nous devons disposer d'instructions capables de ***tester une certaine condition*** et de modifier le comportement du programme en conséquence.

La plus simple de ces instructions conditionnelles est l'instruction **if**. Pour expérimenter son fonctionnement, veuillez entrer dans votre éditeur Python les deux lignes suivantes :

```
>>> a = 150
>>> if (a > 100):
...
```

La première commande affecte la valeur 150 à la variable **a**. Jusqu'ici rien de nouveau.
Lorsque vous finissez d'entrer la seconde ligne, par contre, vous constatez que Python réagit d'une nouvelle manière. En effet, et à moins que vous n'ayez oublié le caractère ":" à la fin de la ligne, vous constatez que le *prompt principal* (>>>) est maintenant remplacé par un *prompt secondaire* constitué de trois points[10].

Si votre éditeur ne le fait pas automatiquement, vous devez à présent effectuer une tabulation (ou entrer 4 espaces) avant d'entrer la ligne suivante, de manière à ce que celle-ci soit ***indentée*** (c.à.d. en retrait) par rapport à la précédente. Votre écran devrait se présenter maintenant comme suit :

```
>>> a = 150
>>> if (a > 100):
...     print "a dépasse la centaine"
...
```

9 Une suite d'instructions est souvent désignée par le terme de ***séquence*** dans les ouvrages qui traitent de la programmation d'une manière générale. Nous préférons réserver ce terme à un concept Python précis, lequel englobe les chaînes de caractères, les tuples et les listes (voir plus loin).

10 Dans certaines versions de l'éditeur Python pour *Window$*, le prompt secondaire n'apparaît pas.

Frappez encore une fois <Enter>. Le programme s'exécute, et vous obtenez :

```
a dépasse la centaine
```

Recommencez le même exercice, mais avec **a = 20** en guise de première ligne : cette fois Python n'affiche plus rien du tout.

L'expression que vous avez placée entre parenthèses est ce que nous appellerons désormais une condition. L'instruction **if** permet de tester la validité de cette condition. Si la condition est vraie, alors l'instruction que nous avons ***indentée*** après le ":" est exécutée. Si la condition est fausse, rien ne se passe. Notez que les parenthèses utilisées ici sont optionnelles sous Python. Nous les avons utilisées pour améliorer la lisibilité. Dans d'autres langages, il se peut qu'elles soient obligatoires.

Recommencez encore, en ajoutant deux lignes comme indiqué ci-dessous. Veillez bien à ce que la quatrième ligne débute tout à fait à gauche (pas d'indentation), mais que la cinquième soit à nouveau indentée (de préférence avec un retrait identique à celui de la troisième) :

```
>>> a = 20
>>> if (a > 100):
...     print "a dépasse la centaine"
... else:
...     print "a ne dépasse pas cent"
...
```

Frappez <Enter> encore une fois. Le programme s'exécute, et affiche cette fois :

```
a ne dépasse pas cent
```

Comme vous l'aurez certainement déjà compris, l'instruction **else** ("sinon", en anglais) permet de programmer une exécution alternative, dans laquelle le programme doit choisir entre deux possibilités. On peut faire mieux encore en utilisant aussi l'instruction **elif** (contraction de "else if") :

```
>>> a = 0
>>> if a > 0 :
...     print "a est positif"
... elif a < 0 :
...     print "a est négatif"
... else:
...     print "a est nul"
...
```

3.3 Opérateurs de comparaison

La condition évaluée après l'instruction if peut contenir les ***opérateurs de comparaison*** suivants :

```
x == y              # x est égal à y
x != y              # x est différent de y
x > y               # x est plus grand que y
x < y               # x est plus petit que y
x >= y              # x est plus grand que, ou égal à y
x <= y              # x est plus petit que, ou égal à y
```

Exemple :

```
>>> a = 7
>>> if (a % 2 == 0):
...     print "a est pair"
...     print "parce que le reste de sa division par 2 est nul"
... else:
...     print "a est impair"
...
```

Notez bien que l'opérateur de comparaison pour l'égalité de deux valeurs est constitué de deux signes "égale" et non d'un seul[11]. (Le signe "égale" utilisé seul est un opérateur d'affectation, et non un opérateur de comparaison. Vous retrouverez le même symbolisme en *C++* et en *Java*).

3.4 Instructions composées – Blocs d'instructions

La construction que vous avez utilisée avec l'instruction **if** est votre premier exemple d'***instruction composée***. Vous en rencontrerez bientôt d'autres. Sous Python, toutes les instructions composées ont toujours la même structure : une ligne d'en-tête terminée par un double point, suivie d'une ou de plusieurs instructions indentées sous cette ligne d'en-tête. Exemple :

```
Ligne d'en-tête:
    première instruction du bloc
    ... ...
    ... ...
    dernière instruction du bloc
```

S'il y a plusieurs instructions indentées sous la ligne d'en-tête, ***elles doivent l'être exactement au même niveau*** (comptez un décalage de 4 caractères, par exemple). Ces instructions indentées constituent ce que nous appellerons désormais un ***bloc d'instructions***. Un bloc d'instructions est une suite d'instructions formant un ensemble logique, qui n'est exécuté que dans certaines conditions définies dans la ligne d'en-tête. Dans l'exemple du paragraphe précédent, les deux lignes d'instructions indentées sous la ligne contenant l'instruction **if** constituent un même bloc logique : ces deux lignes ne sont exécutées - toutes les deux - que si la condition testée avec l'instruction **if** se révèle vraie, c.à.d. si le reste de la division de **a** par 2 est nul.

11 Rappel : l'opérateur % est l'opérateur *modulo* : il calcule le reste d'une division entière. Ainsi par exemple, **a % 2** fournit le reste de la division de **a** par 2.

3.5 Instructions imbriquées

Il est parfaitement possible d'imbriquer les unes dans les autres plusieurs instructions composées, de manière à réaliser des structures de décision complexes. Exemple :

```
if embranchement == "vertébrés":                        # 1
    if classe == "mammifères":                          # 2
        if ordre == "carnivores":                       # 3
            if famille == "félins":                     # 4
                print "c'est peut-être un chat"         # 5
        print "c'est en tous cas un mammifère"          # 6
    elif classe == 'oiseaux':                           # 7
        print "c'est peut-être un canari"               # 8
print"la classification des animaux est complexe"       # 9
```

Analysez cet exemple. Ce fragment de programme n'imprime la phrase "c'est peut-être un chat" que dans le cas où les quatre premières conditions testées sont vraies.

Pour que la phrase "c'est en tous cas un mammifère" soit affichée, il faut et il suffit que les deux premières conditions soient vraies. L'instruction d'affichage de cette phrase (ligne 4) se trouve en effet au même niveau d'indentation que l'instruction : *if ordre == "carnivores":* (ligne 3). Les deux font donc partie d'un même bloc, lequel est entièrement exécuté si les conditions testées aux lignes 1 & 2 sont vraies.

Pour que la phrase "c'est peut-être un canari" soit affichée, il faut que la variable embranchement contienne "vertébrés", et que la variable classe contienne "oiseaux".

Quant à la phrase de la ligne 9, elle est affichée dans tous les cas, parce qu'elle fait partie du même bloc d'instructions que la ligne 1.

3.6 Quelques règles de syntaxe Python

Tout ce qui précède nous amène à faire le point sur quelques règles de syntaxe :

3.6.1 Les limites des instructions et des blocs sont définies par la mise en page

Dans de nombreux langages de programmation, il faut terminer chaque ligne d'instructions par un caractère spécial (souvent le point-virgule). Sous Python, c'est le caractère de fin de ligne[12] qui joue ce rôle. (Nous verrons plus loin comment outrepasser cette règle pour étendre une instruction complexe sur plusieurs lignes). On peut également terminer une ligne d'instructions par un commentaire. Un commentaire Python commence toujours par le caractère spécial # . Tout ce qui est inclus entre ce caractère et le saut à la ligne suivant est complètement ignoré par le compilateur.

Dans la plupart des autres langages, un bloc d'instructions doit être délimité par des symboles spécifiques (parfois même par des instructions, telles que **begin** et **end**). En *C++* et en *Java*, par exemple, un bloc d'instructions doit être délimité par des accolades. Cela permet d'écrire les blocs d'instructions les uns à la suite des autres, sans se préoccuper d'indentation ni de sauts à la ligne, mais cela peut conduire à l'écriture de programmes confus, difficiles à relire pour les pauvres humains que nous sommes. On conseille donc à tous les programmeurs qui utilisent ces langages de se servir ***aussi*** des sauts à la ligne et de l'indentation pour bien délimiter visuellement les blocs.

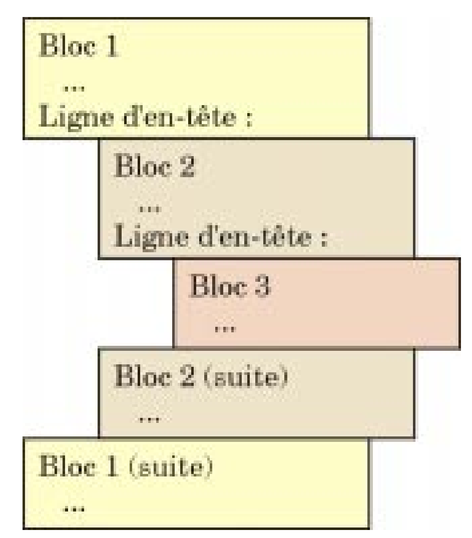
12 Ce caractère n'apparaît ni à l'écran, ni sur les listings imprimés. Il est cependant bien présent, à un point tel qu'il fait même problème dans certains cas, parce qu'il n'est pas encodé de la même manière par tous les systèmes d'exploitation. Nous en reparlerons plus loin, à l'occasion de notre étude des fichiers texte (page 111).

Avec Python, ***vous devez*** utiliser les sauts à la ligne et l'indentation, mais en contrepartie vous n'avez pas à vous préoccuper d'autres symboles délimiteurs de blocs. En définitive, Python vous force donc à écrire du code lisible, et à prendre de bonnes habitudes que vous conserverez lorsque vous utiliserez d'autres langages.

3.6.2 Instruction composée = En-tête , double point , bloc indenté

Nous aurons de nombreuses occasions d'approfondir le concept de "bloc d'instructions" et de faire des exercices à ce sujet, dès le chapitre suivant.

Le schéma ci-contre en résume le principe.



- Les blocs d'instructions sont toujours associés à une ligne d'en-tête contenant une instruction bien spécifique (if, elif, else, while, def, ...) ***se terminant par un double point***.
- ***Les blocs sont délimités par l'indentation*** : toutes les lignes d'un même bloc doivent être indentées exactement de la même manière (c.à.d. décalées vers la droite d'un même nombre d'espaces[13]). Le nombre d'espaces à utiliser pour l'indentation est quelconque, mais la plupart des programmeurs utilisent des multiples de 4.
- Notez que le code du bloc le plus externe (bloc 1) ne peut pas lui-même être écarté de la marge de gauche (Il n'est imbriqué dans rien).

3.6.3 Les espaces et les commentaires sont normalement ignorés

A part ceux qui servent à l'indentation, en début de ligne, les espaces placés à l'intérieur des instructions et des expressions sont presque toujours ignorés (sauf s'ils font partie d'une chaîne de caractères). Il en va de même pour les commentaires : ceux-ci commencent toujours par un caractère dièse (#) et s'étendent jusqu'à la fin de la ligne courante.

13 Vous pouvez aussi indenter à l'aide de tabulations, mais alors vous devrez faire très attention à ne pas utiliser tantôt des espaces, tantôt des tabulations pour indenter les lignes d'un même bloc. En effet, et même si le résultat paraît identique à l'écran, espaces et tabulations sont des codes binaires distincts : Python considérera donc que ces lignes indentées différemment font partie de blocs différents. Il peut en résulter des erreurs difficiles à déboguer.
En conséquence, la plupart des programmeurs préfèrent se passer des tabulations. Si vous utilisez un éditeur "intelligent", vous pouvez escamoter le problème en activant l'option "Remplacer les tabulations par des espaces".

Chapitre 4 : Instructions répétitives.

4.1 Ré-affectation

Nous ne l'avions pas encore signalé explicitement : il est permis de ré-affecter une nouvelle valeur à une même variable, autant de fois qu'on le souhaite.

L'effet d'une ré-affectation est de remplacer l'ancienne valeur d'une variable par une nouvelle.

```
>>> altitude = 320
>>> print altitude
320
>>> altitude = 375
>>> print altitude
375
```

Ceci nous amène à attirer une nouvelle fois votre attention sur le fait que le symbole *égale* utilisé sous Python pour réaliser une affectation ne doit en aucun cas être confondu avec un symbole d'égalité tel qu'il est compris en mathématique. Il est tentant d'interpréter l'instruction **altitude = 320** comme une affirmation d'égalité, mais ce n'en n'est pas une !

- Premièrement, l'égalité est commutative, alors que l'affectation ne l'est pas. Ainsi, en mathématique, les écritures **a = 7** et **7 = a** sont équivalentes, alors qu'une instruction de programmation telle que **375 = altitude** serait illégale.
- Deuxièmement, l'égalité est permanente, alors que l'affectation peut être remplacée comme nous venons de le voir. Lorsqu'en mathématique, nous affirmons une égalité telle que **a = b** au début d'un raisonnement, alors **a** continue à être égal à **b** durant tout le développement qui suit.
  En programmation, une première instruction d'affectation peut rendre égales les valeurs de deux variables, et une instruction ultérieure en changer ensuite l'une ou l'autre. Exemple :

```
>>> a = 5
>>> b = a                   # a et b contiennent des valeurs égales
>>> b = 2                   # a et b sont maintenant différentes
```

Rappelons ici que Python permet d'affecter leurs valeurs à plusieurs variables simultanément :

```
>>> a, b, c, d = 3, 4, 5, 7
```

Cette fonctionnalité de Python est bien plus intéressante encore qu'elle n'en a l'air à première vue. Supposons par exemple que nous voulions maintenant échanger les valeurs des variables **a** et **c**. (Actuellement, **a** contient la valeur 3, et **c** la valeur 5. Nous voudrions que ce soit l'inverse). Comment faire ?

Exercice :

e 4. Écrivez les lignes d'instructions nécessaires pour obtenir ce résultat.

A la suite de l'exercice proposé ci-dessus, vous aurez certainement trouvé une méthode, et votre professeur vous demandera probablement de la commenter en classe. Comme il s'agit d'une opération courante, les langages de programmation proposent souvent des raccourcis pour l'effectuer (par exemple des instructions spécialisées, telle l'instruction SWAP du langage *Basic*). Sous Python, *l'affectation multiple* permet de programmer l'échange d'une manière particulièrement élégante :

```
>>> a, b = b, a
```

(On pourrait bien entendu échanger d'autres variables en même temps, dans la même instruction).

4.2 Répétitions en boucle - l'instruction while

L'une des choses que les machines font le mieux est la répétition sans erreur de tâches identiques. Il existe bien des méthodes pour programmer ces tâches répétitives. Nous allons commencer par l'une des plus fondamentales : la boucle construite à partir de l'instruction **while**.

Veuillez donc entrer les commandes ci-dessous :

```
>>> a = 0
>>> while (a < 7):                 # (n'oubliez pas le double point !)
...     a = a + 1                  # (n'oubliez pas l'indentation !)
...     print a
```

Frappez encore une fois <Enter>.

Que se passe-t-il ?

Avant de lire les commentaires de la page suivante, prenez le temps d'ouvrir votre cahier et d'y noter cette série de commandes. Décrivez aussi le résultat obtenu, et essayez de l'expliquer de la manière la plus détaillée possible.

Commentaires

Le mot **while** signifie "tant que" en anglais. Cette instruction utilisée à la seconde ligne indique à Python qu'il lui faut ***répéter continuellement le bloc d'instructions qui suit, tant que*** le contenu de la variable **a** reste inférieur à 7.

Comme l'instruction **if** abordée au chapitre précédent, l'instruction **while** amorce une ***instruction composée***. Le double point à la fin de la ligne introduit le bloc d'instructions à répéter, lequel doit obligatoirement se trouver en retrait. Comme vous l'avez appris au chapitre précédent, toutes les instructions d'un même bloc doivent être indentées exactement au même niveau (c.à.d. décalées à droite d'un même nombre d'espaces).

Nous avons ainsi construit notre première ***boucle de programmation***, laquelle répète un certain nombre de fois le bloc d'instructions indentées. Voici comment cela fonctionne :

- Avec l'instruction **while**, Python commence par évaluer la validité de la condition fournie entre parenthèses (Celles-ci sont optionnelles. Nous ne les avons utilisées que pour clarifier notre explication).
- Si la condition se révèle fausse, alors tout le bloc qui suit est ignoré et l'exécution du programme se termine[14].
- Si la condition est vraie, alors Python exécute tout le bloc d'instructions constituant ***le corps de la boucle***, c.à.d. :
    - l'instruction **a = a + 1** qui *incrémente* d'une unité le contenu de la variable **a** (c.à.d. que l'on affecte à la la variable **a** une nouvelle valeur, qui est égale à la valeur précédente augmentée d'une unité).
    - l'instruction **print** qui affiche la valeur courante de la variable **a**
- lorsque ces deux instructions ont été exécutées, nous avons assisté à une première **itération**, et le programme boucle, c.à.d. que l'exécution reprend à la ligne contenant l'instruction **while**. La condition qui s'y trouve est à nouveau évaluée, et ainsi de suite.
  Dans notre exemple, si la *condition* **a < 7** est encore vraie, le corps de la boucle est exécuté une nouvelle fois et le bouclage se poursuit.

Remarques :

- La variable évaluée dans la condition doit exister au préalable (Il faut qu'on lui ait déjà affecté au moins une valeur)
- Si la condition est fausse au départ, le corps de la boucle n'est jamais exécuté
- Si la condition reste toujours vraie, alors le corps de la boucle est répété indéfiniment (tout au moins tant que Python lui-même continue à fonctionner). Il faut donc veiller à ce que le corps de la boucle contienne au moins une instruction qui change la valeur d'une variable intervenant dans la condition évaluée par **while**, de manière à ce que cette condition puisse devenir fausse et la boucle se terminer.
  Exemple de boucle sans fin (à éviter) :

```
>>> n = 3
>>> while n < 5:
...     print "hello !"
```

14 ... du moins dans cet exemple. Nous verrons un peu plus loin qu'en fait l'exécution continue avec la première instruction qui suit le bloc indenté, et qui fait partie du même bloc que l'instruction while elle-même.

4.3 Élaboration de tables

Recommencez à présent le premier exercice, mais avec la petite modification ci-dessous :

```
>>> a = 0
>>> while a < 12:
...     a = a +1
...     print a , a**2 , a**3
```

Vous devriez obtenir la liste des carrés et des cubes des nombres de 1 à 12.
Notez au passage que l'instruction **print** permet d'afficher plusieurs expressions l'une à la suite de l'autre sur la même ligne : il suffit de les séparer par des virgules. Python insère automatiquement un espace entre les éléments affichés.

4.4 Construction d'une suite mathématique

Le petit programme ci-dessous permet d'afficher les dix premiers termes d'une suite appelée "Suite de *Fibonacci*". Il s'agit d'une suite de nombres, dont chaque terme est égal à la somme des deux termes qui le précèdent. Analysez ce programme (qui utilise judicieusement l'*affectation multiple*). Décrivez le mieux possible le rôle de chacune des instructions.

```
>>> a, b, c = 1, 1, 1
>>> while c < 11 :
...    print b,
...    a, b, c = b, a+b, c+1
```

Lorsque vous lancez l'exécution de ce programme, vous obtenez :

```
1 2 3 5 8 13 21 34 55 89
```

Les termes de la suite de *Fibonacci* sont affichés sur la même ligne. Vous obtenez ce résultat grâce à la virgule placée à la fin de la ligne qui contient l'instruction **print**. Si vous supprimez cette virgule, les nombres seront affichés l'un en-dessous de l'autre.

Exercices :

e 5. Ecrivez un programme qui affiche les 20 premiers termes de la table de multiplication par 7.

e 6. Ecrivez un programme qui affiche une table de conversion de sommes d'argent exprimées en euros, en dollars canadiens. La progression des sommes de la table sera "géométrique", comme dans l'exemple ci-dessous :

```
1 euro(s) = 1.65 dollar(s)
2 euro(s) = 3.30 dollar(s)
4 euro(s) = 6.60 dollar(s)
8 euro(s) = 13.20 dollar(s)
```

etc. (S'arrêter à 16384 euros)

e 7. Ecrivez un programme qui affiche une suite de 12 nombres dont chaque terme soit égal au triple du terme précédent.

4.5 Premiers scripts, ou : Comment conserver nos programmes ?

Jusqu'à présent, vous avez toujours utilisé Python *en mode interactif* (c.à.d. que vous avez à chaque fois entré les commandes directement dans l'interpréteur, sans les sauvegarder au préalable dans un fichier). Cela vous a permis d'apprendre très rapidement les bases du langage, par expérimentation directe. Cette façon de faire présente toutefois un gros inconvénient : toutes les séquences d'instructions que vous avez écrites disparaissent irrémédiablement dès que vous fermez l'interpréteur. Avant de poursuivre plus avant votre étude, il est donc temps que vous appreniez à sauvegarder vos programmes dans des fichiers, sur disque dur ou disquette, de manière à pouvoir les retravailler par étapes successives, les transférer sur d'autres machines, etc.

Pour ce faire, vous allez désormais rédiger vos séquences d'instructions dans un ***éditeur de textes*** quelconque (par exemple *Joe*, *Nedit, Kate* ... sous *Linux, Edit* sous *MS-DOS*, *Wordpad* sous *Window$*, ou mieux encore l'éditeur incorporé dans une interface de développement telle que *IDLE* ou *PythonWin).* Ainsi vous écrirez un ***script***, que vous pourrez ensuite sauvegarder, modifier, copier, etc. comme n'importe quel autre texte traité par ordinateur[15].

La figure ci-dessous illustre l'utilisation de l'éditeur *Nedit* sous *Gnome (Linux)* :

fibo1.py - /dos_data/python/essais/

File Edit Search Preferences Shell Macro Windows Help

/dos_data/python/essais/fibo1.py DOS 496 bytes

```
# Premier essai de script Python
# petit programme simple affichant une suite de Fibonacci, c.à.d. une suite
# de nombres dont chaque terme est égal à la somme des deux précédents.

print "Suite de Fibonacci :"

a,b,c = 1,1,1                # a & b servent au calcul des termes successifs
                             # c est un simple compteur
print 1                      # affichage du premier terme
while c<15:                  # nous afficherons 15 termes au total
    a,b,c = b,a+b,c+1
    print b
```

Par la suite, lorsque vous voudrez tester l'exécution de votre programme, il vous suffira de lancer l'interpréteur Python en lui fournissant (comme argument) le nom du fichier qui contient le script. Par exemple, si vous avez placé un script dans un fichier nommé "MonScript", il suffira d'entrer la commande suivante dans une fenêtre de terminal pour que ce script s'exécute :

```
python MonScript
```

Pour faire mieux encore, veillez à donner au fichier un nom qui se termine par l'extension **.py**

Si vous respectez cette convention, vous pourrez (sous *Window$, KDE, Gnome, ...*) lancer l'exécution du script, simplement en cliquant sur son nom ou sur l'icône correspondante dans le gestionnaire de fichiers (c.à.d. l'explorateur, sous *Window$*, ou *Konqueror*, sous *KDE*).

15 Il serait parfaitement possible d'utiliser un système de traitement de textes, à la condition d'effectuer la sauvegarde sous un format "texte pur" (sans balises de mise en page). Il est cependant préférable d'utiliser un véritable éditeur ANSI "intelligent" tel que *nedit* ou *IDLE*, muni d'une fonction de coloration syntaxique pour Python, qui vous aide à éviter les fautes de syntaxe. Avec *IDLE*, suivez le menu : File → New window (ou frappez CTRL-N) pour ouvrir une nouvelle fenêtre dans laquelle vous écrirez votre script. Pour l'exécuter, il vous suffira (après sauvegarde), de suivre le menu : Edit → Run script (ou de frapper CTRL-F5).

Ces gestionnaires graphiques "savent" en effet qu'il doivent lancer l'interpréteur Python chaque fois que leur utilisateur essaye d'ouvrir un fichier dont le nom se termine par **.py**. (Cela suppose bien entendu qu'ils aient été correctement configurés). La même convention permet en outre aux éditeurs "intelligents" de reconnaître automatiquement les scripts Python et d'adapter leur coloration syntaxique en conséquence.

Un script Python contiendra des séquences d'instructions identiques à celles que vous avez expérimentées jusqu'à présent. Puisque ces séquences sont destinées à être conservées et relues plus tard par vous-même ou par d'autres, ***il vous est très fortement recommandé d'expliciter vos scripts le mieux possible, en y incorporant de nombreux commentaires.*** La principale difficulté de la programmation consiste en effet à mettre au point des algorithmes corrects. Afin que ces algorithmes puissent être vérifiés, corrigés, modifiés, etc. dans de bonnes conditions, il est essentiel que leur auteur les décrive le plus complètement et le plus clairement possible. Et le meilleur emplacement pour cette description est le corps même du script (ainsi elle ne peut pas s'égarer).

Un bon programmeur veille toujours à insérer un grand nombre de commentaires dans ses scripts. En procédant ainsi, non seulement il facilite la compréhension de ses algorithmes pour d'autres lecteurs éventuels, mais encore il se force lui-même à avoir les idées plus claires.

On peut insérer des commentaires quelconques à peu près n'importe où dans un script. Il suffit de les faire précéder d'un caractère #. Lorsqu'il rencontre ce caractère, l'interpréteur Python ignore tout ce qui suit, jusqu'à la fin de la ligne courante.

Comprenez bien qu'il est important d'inclure des commentaires ***au fur et à mesure*** de l'avancement de votre travail de programmation. N'attendez pas que votre script soit terminé pour les ajouter "après coup". Vous vous rendrez progressivement compte qu'un programmeur passe énormément de temps à relire son propre code (pour le modifier, y rechercher des erreurs, etc). Cette relecture sera grandement facilitée si le code comporte de nombreuses explications et remarques.

Ouvrez donc un éditeur de texte, et rédigez le script ci-dessous :

```
# Premier essai de script Python
# petit programme simple affichant une suite de Fibonacci, c.à.d. une suite
# de nombres dont chaque terme est égal à la somme des deux précédents.

a, b, c = 1, 1, 1            # a & b servent au calcul des termes successifs
                             # c est un simple compteur
print 1                      # affichage du premier terme
while c<15:                  # nous afficherons 15 termes au total
    a, b, c = b, a+b, c+1
    print b
```

Afin de vous montrer tout de suite le bon exemple, nous commençons ce script par trois lignes de commentaires, qui contiennent une courte description de la fonctionnalité du programme. Prenez l'habitude de faire de même dans vos propres scripts.

Les lignes de code elle-mêmes sont documentées. Si vous procédez comme nous l'avons fait, c.à.d. en insérant des commentaires à la droite des instructions correspondantes, veillez à les écarter suffisamment de celles-ci, afin de ne pas gêner leur lisibilité.

Lorsque vous aurez bien vérifié votre texte, sauvegardez-le et exécutez-le.

Note : Bien que ce ne soit pas indispensable, nous vous recommandons une fois encore de choisir pour vos scripts des noms de fichiers se terminant par l'extension **.py** Cela aide beaucoup à les identifier comme tels dans un répertoire. Les gestionnaires graphiques de fichiers (*explorateur Window$*, *Konqueror*) se servent d'ailleurs de cette extension pour leur associer une icône spécifique. Evitez cependant de choisir des noms qui risqueraient d'être déjà attribués à des modules python existants : des noms tels que *math.py* ou *Tkinter.py*, par exemple, sont à proscrire !

Si vous travaillez en mode texte sous *Linux*, ou dans une fenêtre *M$DOS*, vous pouvez exécuter votre script à l'aide de la commande **python** suivie du nom du script. Si vous travaillez en mode graphique sous *Linux*, vous pouvez ouvrir une fenêtre de terminal et faire la même chose. Dans l'*explorateur Window$* ou dans *Konqueror*, vous pouvez lancer l'exécution de votre script en effectuant un simple clic de souris sur l'icône correspondante.

Si vous travaillez avec *IDLE*, vous pouvez lancer l'exécution du script en cours d'édition, directement à l'aide de la combinaison de touches <Ctrl-F5>. Consultez votre professeur concernant les autres possibilités de lancement éventuelles sur différents systèmes d'exploitation.

Exercices :

e 8. Ecrivez un programme qui calcule le volume d'un parallélépipède rectangle dont sont fournis au départ la largeur, la hauteur et la profondeur.

e 9. Ecrivez un programme qui convertisse un nombre entier de secondes fourni au départ, en un nombre d'années, de mois, de jours, de minutes et de secondes.
(Utilisez l'opérateur modulo : %).

e 10. Ecrivez un programme qui affiche les 20 premiers termes de la table de multiplication par 7, en signalant au passage (à l'aide d'une astérisque) ceux qui sont des multiples de 3.
Exemple : 7 14 21 * 28 35 42 * 49

e 11. Ecrivez un programme qui calcule les 50 premiers termes de la table de multiplication par 13, mais n'affiche que ceux qui sont des multiples de 7.

e 12. Ecrivez un programme qui affiche la suite de symboles suivante :

```
*
**
***
****
*****
******
*******
********
```

Chapitre 5 : Principaux types de données

5.1 Les données numériques

Dans les exercices réalisés jusqu'à présent, nous avons déjà utilisé des données de deux types : les nombres ***entiers*** ordinaires et les nombres ***réels*** (aussi appelés nombres *à virgule flottante*). Tâchons de mettre en évidence les caractéristiques (et les limites) de ces concepts :

5.1.1 Les types "integer" et "long"

Supposons que nous voulions modifier légèrement notre précédent exercice sur la suite de *Fibonacci*, de manière à obtenir l'affichage d'un plus grand nombre de termes. *A priori*, il suffit de modifier la condition de bouclage, dans la deuxième ligne. Avec "**`while c<49:`**" , nous devrions obtenir quarante-huit termes. Modifions donc légèrement l'exercice, de manière à afficher aussi le type de la variable principale :

```
>>> a, b, c = 1, 1, 1
>>> while c<49:
        print c, " : ", b, type(b)
        a, b, c = b, a+b, c+1
...
...
...  (affichage des 43 premiers termes)
...
44  :  1134903170 <type 'int'>
45  :  1836311903 <type 'int'>
46  :  2971215073 <type 'long'>
47  :  4807526976 <type 'long'>
48  :  7778742049 <type 'long'>
```

Que pouvons-nous constater ?

Si nous n'avions pas utilisé la fonction **type()**, qui nous permet de vérifier à chaque itération le type de la variable **b**, nous n'aurions rien remarqué du tout : la suite des nombres de Fibonacci s'affiche sans problème (et nous pourrions encore l'allonger de nombreux termes supplémentaires).

Il semble donc que Python soit capable de traiter des nombres entiers de taille illimitée.

L'exercice que nous venons de réaliser indique cependant qu'il se passe "quelque chose" lorsque ces nombres deviennent très grands. Au début du programme, les variables **a**, **b** et **c** sont définies implicitement comme étant du type ***integer***. C'est ce qui se passe toujours avec Python lorsqu'on affecte une valeur entière à une variable, à condition que cette valeur ne soit pas trop grande. Dans la mémoire de l'ordinateur, ce type de donnée est en effet encodé sous la forme d'un bloc de 4 octets (ou 32 bits). Or la gamme de valeurs décimales qu'il est possible d'encoder sur 4 octets seulement s'étend de -2147483648 à + 2147483647 (Voir cours d'informatique générale).

Les calculs effectués avec ce type de variable sont toujours très rapides, parce que le processeur de l'ordinateur est capable de traiter directement par lui-même de tels nombres entiers à 32 bits. Lorsqu'il est question de traiter des nombres entiers plus grands, par contre, ou encore des nombres réels (nombres "à virgule flottante"), les logiciels que sont les interpréteurs et compilateurs doivent effectuer un gros travail de codage/décodage, afin de ne présenter en définitive au processeur que des opérations binaires sur des nombres entiers de 32 bits au maximum.

Vous savez déjà que le type des variables Python est défini de manière dynamique.

Puisqu'il s'agit du type le plus performant (aussi bien en termes de vitesse de calcul qu'en termes d'occupation de place dans la mémoire), Python utilise le type ***integer*** par défaut, chaque fois que cela est possible, c.à.d. tant que les valeurs traitées sont des entiers compris entre les limitées déjà mentionnées plus haut (environ 2 milliards, en positif ou en négatif).

Lorsque les valeurs traitées sont des nombres entiers se situant au-delà de ces limites, leur encodage dans la mémoire de l'ordinateur devient plus complexe. Les variables auxquelles on affecte de tels nombres sont alors automatiquement définies comme appartenant au type "entier long" (lequel est désigné par ***long*** dans la terminologie Python).

Ce type ***long*** permet l'encodage de valeurs entières avec une précision quasi infinie : une valeur définie sous cette forme peut en effet posséder un nombre de chiffres significatifs quelconque, ce nombre ***n'étant limité que par la taille de la mémoire disponible sur l'ordinateur utilisé*** !

Exemple :

```
>>> a, b, c = 3, 2, 1
>>> while c < 15:
        print c, ": ", b
        a, b, c = b, a*b, c+1

1 :  2
2 :  6
3 :  12
4 :  72
5 :  864
6 :  62208
7 :  53747712
8 :  3343537668096
9 :  179707499645975396352
10 :  600858794305667322270155425185792
11 :  107978831564966913814384922944738457859243070439030784
12 :  648800305446607527907368373691049776950010342842280428918276494561862345
82611607420928
13 :  700566989011183200292376413995762169216245450579726979173836923132717548
83621235064434673400268965204696103008832506249008437424702378475552
14 :  454528076456265799856362940482493512051682398707229461514016556556583986
42227616335815123825782460196980206141536747116094173550514227947953005917009
6950422693079038247634055829175296831946224503933501754776033004012758368256
>>>
```

Dans l'exemple ci-dessus, la valeur des nombres affichés augmente très rapidement, car chacun d'eux est égal au produit des deux termes précédents.

Au départ, les variables **a**, **b** et **c** sont du type ***integer***, puisqu'on leur affecte des petites valeurs numériques entières : 3, 2 et 1. A partir de la 8e itération, cependant, les variables **b** et **a** sont automatiquement converties l'une après l'autre dans le type ***long*** : le résultat de la multiplication des termes 6 et 7 est en effet déjà bien supérieur à la limite des 2 milliards évoquée plus haut.

La progression continue avec des nombres de plus en plus gigantesques, mais la vitesse de calcul diminue. Les nombres mémorisés sous le type ***long*** occupent une place variable dans la mémoire de l'ordinateur, en fonction de leur taille.

5.1.2 Le type "float"

Vous avez déjà rencontré précédemment cet autre type de donnée numérique : le type "nombre réel", ou "nombre à virgule flottante", désigné en anglais par l'expression "*floating point number*", et que pour cette raison on appellera type ***float*** sous Python.

Ce type autorise les calculs sur de très grands ou très petits nombres (données scientifiques, par exemple), avec un degré de précision constant.

Pour qu'une donnée numérique soit considérée par Python comme étant du type ***float***, il suffit qu'elle contienne dans sa formulation un élément tel qu'un point décimal ou un exposant de 10.

Par exemple, les données :

```
3.14      10.      .001      1e100      3.14e-10
```

sont automatiquement interprétées par Python comme étant du type ***float***.

Essayons donc ce type de données dans un nouveau petit programme (inspiré du précédent) :

```
>>> a, b, c = 1., 2., 1            # => a et b seront du type 'float'
>>> while c <18:
...     a, b, c = b, b*a, c+1
...     print b

2.0
4.0
8.0
32.0
256.0
8192.0
2097152.0
17179869184.0
3.6028797019e+16
6.18970019643e+26
2.23007451985e+43
1.38034926936e+70
3.07828173409e+113
4.24910394253e+183
1.30799390526e+297
         Inf
         Inf
```

Comme vous l'aurez certainement bien compris, nous affichons cette fois encore une série dont les termes augmentent extrêmement vite, chacun d'eux étant égal au produit des deux précédents. Au huitième terme, nous dépassons déjà largement la capacité d'un *integer*. Au neuvième terme, Python passe automatiquement à la notation scientifique ("e+n" signifie en fait : "fois dix à l'exposant n"). Après le quinzième terme, nous assistons à nouveau à un dépassement de capacité (sans message d'erreur) : les nombres vraiment trop grands sont tout simplement notés "inf" (pour "infini").

Le type ***float*** utilisé dans notre exemple permet de manipuler des nombres (positifs ou négatifs) compris entre 10^{-308} et 10^{308} avec une précision de 12 chiffres significatifs. Ces nombres sont encodés d'une manière particulière sur 8 octets (64 bits) dans la mémoire de la machine : une partie du code correspond aux 12 chiffres significatifs, et une autre à l'ordre de grandeur (exposant de 10).

Exercices :

e 13. Écrivez un programme qui convertisse en radians un angle fourni au départ en degrés, minutes, secondes.

e 14. Écrivez un programme qui convertisse en degrés, minutes, secondes un angle fourni au départ en radians.

e 15. Écrivez un programme qui convertisse en degrés Celsius une température exprimée au départ en degrés Fahrenheit, ou l'inverse.
La formule de conversion est : $T_F = T_C \times 1{,}8 + 32$.

e 16. Écrivez un programme qui calcule les intérêts accumulés chaque année pendant 20 ans, par capitalisation d'une somme de 100 euros placée en banque au taux fixe de 4,3 %

e 17. Une légende de l'Inde ancienne raconte que le jeu d'échecs a été inventé par un vieux sage, que son roi voulut remercier en lui affirmant qu'il lui accorderait n'importe quel cadeau en récompense. Le vieux sage demanda qu'on lui fournisse simplement un peu de riz pour ses vieux jours, et plus précisément un nombre de grains de riz suffisant pour que l'on puisse en déposer 1 seul sur la première case du jeu qu'il venait d'inventer, deux sur la suivante, quatre sur la troisième, et ainsi de suite jusqu'à la 64e case.
Écrivez un programme Python qui affiche le nombre de grains à déposer sur chacune des 64 cases du jeu. Calculez ce nombre de deux manières :
- le nombre exact de grains (nombre entier)
- le nombre de grains en notation scientifique (nombre réel)

5.2 Les données alphanumériques

Jusqu'à présent nous n'avons manipulé que des nombres. Mais un programme d'ordinateur peut également traiter des caractères alphabétiques, des mots, des phrases, ou des suites de symboles quelconques. Dans la plupart des langages de programmation, il existe pour cet usage une structure de données que l'on appelle ***chaîne de caractères*** (ou ***string*** en anglais).

5.2.1 Le type "string" (chaîne de caractères)

Sous Python, une donnée de type ***string*** est une suite quelconque de caractères délimitée soit par des apostrophes (*simple quotes*), soit par des guillemets (*double quotes*).

Exemples :

```
>>> phrase1 = 'les oeufs durs.'
>>> phrase2 = '"Oui", répondit-il,'
>>> phrase3 = "j'aime bien"
>>> print phrase2, phrase3, phrase1
"Oui", répondit-il, j'aime bien les oeufs durs.
```

Les 3 variables **phrase1**, **phrase2**, **phrase3** sont donc des variables de type ***string***.

Remarquez l'utilisation des guillemets pour délimiter une chaîne dans laquelle il y a des apostrophes, ou l'utilisation d'apostrophes pour délimiter une chaîne qui contient des guillemets. Remarquez aussi encore une fois que l'instruction **print** insère un espace entre les éléments affichés.

Le caractère spécial "\" (*antislash*) permet quelques subtilités complémentaires :

- En premier lieu, il permet d'écrire sur plusieurs lignes une commande qui serait trop longue pour tenir sur une seule (cela vaut pour n'importe quel type de commande).
- A l'intérieur d'une chaîne de caractères, l'*antislash* permet d'insérer un certain nombre de codes spéciaux (sauts à la ligne, apostrophes, guillemets, etc.). Exemples :

```
>>> txt3 = '"N\'est-ce pas ?" répondit-elle.'
>>> print txt3
"N'est-ce pas ?" répondit-elle.
>>> Salut = "Ceci est une chaîne plutôt longue\n contenant plusieurs lignes \
... de texte (Ceci fonctionne\n de la même façon en C/C++.\n\
...     Notez que les blancs en début\n de ligne sont significatifs.\n"
>>> print Salut
Ceci est une chaîne plutôt longue
 contenant plusieurs lignes de texte (Ceci fonctionne
 de la même façon en C/C++.
    Notez que les blancs en début
 de ligne sont significatifs.
```

Remarques :

- La séquence **\n** dans une chaîne provoque un saut à la ligne.
- La séquence **\'** permet d'insérer une apostrophe dans une chaîne délimitée par des apostrophes
- Rappelons encore ici que la casse est significative dans les noms de variables (Il faut respecter scrupuleusement le choix initial de majuscules ou minuscules).

***"Triple quotes"* :**

Pour insérer plus aisément des caractères spéciaux ou "exotiques" dans une chaîne, sans faire usage de l'*antislash*, ou pour faire accepter l'*antislash* lui-même dans la chaîne, on peut encore délimiter la chaîne à l'aide de ***triples guillemets*** ou de ***triples apostrophes*** :

```
>>> a1 = """
... Usage: trucmuche[OPTIONS]
... { -h
...    -H hôte
... }"""

>>> print a1

Usage: trucmuche[OPTIONS]
{ -h
  -H hôte
}
```

5.2.2 Accès aux caractères individuels d'une chaîne

Les chaînes de caractères constituent un cas particulier d'un type de données plus général que l'on appelle des ***données composites***. Une donnée composite est une entité qui rassemble dans une seule structure un ensemble d'entités plus simples : dans le cas d'une chaîne de caractères, par exemple, ces entités plus simples sont évidemment les caractères eux-mêmes. En fonction des circonstances, nous souhaiterons traiter la chaîne de caractères, tantôt comme un seul objet, tantôt comme une collection de caractères distincts. Un langage de programmation tel que Python doit donc être pourvu de mécanismes qui permettent d'accéder séparément à chacun des caractères d'une chaîne. Comme vous allez le voir, cela n'est pas bien compliqué :

Python considère qu'une chaîne de caractères est un objet de la catégorie des ***séquences***, lesquelles sont des ***collections ordonnées d'éléments***. Ce que l'on faire remarquer ainsi est le fait que les caractères de la chaîne sont toujours disposés dans un certain ordre. Chaque caractère d'une chaîne peut donc être désigné par sa place dans la séquence, à l'aide d'un index.

Pour accéder à un caractère bien déterminé, on utilise le nom de la variable qui contient la chaîne, et on lui accole entre deux crochets l'index numérique qui correspond à la position du caractère dans la chaîne.

Attention, cependant : comme vous aurez l'occasion de le vérifier par ailleurs, les données informatiques sont presque toujours numérotées ***à partir de zéro*** (et non à partir de un). C'est le cas pour les caractères d'une chaîne.

Exemple :

```
>>> ch = "Stéphanie"
>>> print ch[0], ch[3]
S p
```

5.2.3 Opérations élémentaires sur les chaînes

Python intègre de nombreuses ***fonctions*** qui permettent d'effectuer divers traitements sur les chaînes de caractères (conversions majuscules/minuscules, découpage en chaînes plus petites, recherche de mots, etc.). Nous approfondirons ce sujet un peu plus loin (voir page 117).

Pour l'instant, nous pouvons nous contenter de savoir qu'il est possible d'accéder individuellement à chacun des caractères d'une chaîne, comme cela a été expliqué ci-dessus. Sachons en outre que l'on peut aussi :

- assembler plusieurs petites chaînes pour en construire de plus grandes. Cette opération s'appelle concaténation et on la réalise sous Python à l'aide de l'opérateur + (Cet opérateur réalise donc l'opération d'addition lorsqu'on l'applique à des nombres, et l'opération de concaténation lorsqu'on l'applique à des chaînes de caractères. Exemple :

```
a = 'Petit poisson'
b = ' deviendra grand'
c = a + b
print c
petit poisson deviendra grand
```

- déterminer la longueur (c.à.d. le nombre de caractères) d'une chaîne, en faisant appel à la fonction intégrée **len()** :

```
>>> print len(c)
29
```

- Convertir en nombre véritable une chaîne de caractères qui représente un nombre. Exemple :

```
>>> ch = '8647'
>>> print ch + 45
   ==> *** erreur *** on ne peut pas additionner une chaîne et un nombre
>>> n = int(ch)
>>> print n + 65
8712                        # OK : on peut additionner 2 nombres
```

Dans cet exemple, la fonction intégrée **int()** convertit la chaîne en nombre entier. Il serait également possible de convertir une chaîne en nombre réel à l'aide de la fonction **float()**.

Exercices :

e 18. Écrivez un script qui détermine si une chaîne contient ou non le caractère "e".

e 19. Écrivez un script qui compte le nombre d'occurrences du caractère "e" dans une chaîne.

e 20. Écrivez un script qui recopie une chaîne (dans une nouvelle variable), en insérant des astérisques entre les caractères.
Ainsi par exemple, **`"gaston"`** devra devenir **`"g*a*s*t*o*n"`**

e 21. Écrivez un script qui recopie une chaîne (dans une nouvelle variable) en l'inversant.
Ainsi par exemple, **`"zorglub"`** deviendra **`"bulgroz"`**.

e 22. En partant de l'exercice précédent, écrivez un script qui détermine si une chaîne de caractères donnée est un palindrome (c.à.d. une chaîne qui peut se lire indifféremment dans les deux sens), comme par exemple "radar" ou "sos".

5.3 Les listes (première approche)

Les chaînes que nous avons abordées à la rubrique précédente constituaient un premier exemple de ***données composites***, lesquelles sont utilisées pour regrouper de manière structurée des ensembles de valeurs. Vous apprendrez progressivement à utiliser plusieurs autres types de données composites, parmi lesquelles les ***listes***, les ***tuples*** et les ***dictionnaires***.[16] Nous n'allons cependant aborder ici que le premier de ces trois types, et ce de façon assez sommaire. Il s'agit là en effet d'un sujet fort vaste, sur lequel nous devrons revenir à plusieurs reprises.

Sous Python, on peut définir une liste comme ***une collection d'éléments séparés par des virgules, l'ensemble étant enfermé dans des crochets.*** Exemple :

```
>>> jour = ['lundi', 'mardi', 'mercredi', 1800, 20.357, 'jeudi', 'vendredi']
>>> print jour
['lundi', 'mardi', 'mercredi', 1800, 20.357, 'jeudi', 'vendredi']
```

Dans cet exemple, la valeur de la variable **jour** est une liste.

Comme on peut le constater dans l'exemple choisi, les éléments qui constituent une liste peuvent être de types variés. Dans cet exemple, en effet, les trois premiers éléments sont des chaînes de caractères, le quatrième élément est un entier, le cinquième un réel, etc. (Nous verrons plus loin qu'un élément d'une liste peut lui-même être une liste !). A cet égard, le concept de liste est donc assez différent du concept de "tableau" ou de "variable indicée" que l'on rencontre dans d'autres langages de programmation.

Remarquons aussi que comme les chaînes de caractères, les listes sont des ***séquences***, c.à.d. des collections ***ordonnées*** d'objets. Les divers éléments qui constituent une liste sont en effet toujours disposés dans le même ordre, et l'on peut donc accéder à chacun d'entre eux individuellement si l'on connaît son index dans la liste. Comme c'était déjà le cas pour les caractères dans une chaîne, il faut cependant retenir que la numérotation de ces index commence à partir de zéro, et non à partir de un.

Exemples :

```
>>> jour = ['lundi', 'mardi', 'mercredi', 1800, 20.357, 'jeudi', 'vendredi']
>>> print jour[2]
mercredi
>>> print jour[4]
20.357
```

A la différence de ce qui se passe pour les chaînes, qui constituent un type de données non-modifiables (nous aurons plus loin diverses occasions de revenir là-dessus), il est possible de changer les éléments individuels d'une liste :

```
>>> print jour
['lundi', 'mardi', 'mercredi', 1800, 20.357, 'jeudi', 'vendredi']
>>> jour[3] = jour[3] +47
>>> print jour
['lundi', 'mardi', 'mercredi', 1847, 20.357, 'jeudi', 'vendredi']
```

16 Vous pourrez même créer vos propres types de données composites, lorsque vous aurez assimilé le concept de ***classe*** (voir page 145).

On peut donc remplacer certains éléments d'une liste par d'autres, comme ci-dessous :

```
>>> jour[3] = 'Juillet'
>>> print jour
['lundi', 'mardi', 'mercredi', 'Juillet', 20.357, 'jeudi', 'vendredi']
```

La ***fonction intégrée*** **len()** , que nous avons déjà rencontrée à propos des chaînes, s'applique aussi aux listes. Elle renvoie le nombre d'éléments présents dans la liste :

```
>>> len(jour)
7
```

Une autre ***fonction intégrée*** permet de supprimer d'une liste un élément quelconque (à partir de son index). Il s'agit de la fonction **del()** [17] :

```
>>> del(jour[4])
>>> print jour
['lundi', 'mardi', 'mercredi', 'juillet', 'jeudi', 'vendredi']
```

Il est également tout à fait possible d'ajouter un élément à une liste, mais pour ce faire, il faut considérer que la liste est un ***objet***, dont on va utiliser l'une des ***méthodes***. Les concepts informatiques d'***objet*** et de ***méthode*** ne seront expliqués qu'un peu plus loin dans ces notes, mais nous pouvons dès à présent montrer "comment ça marche" dans le cas particulier d'une liste :

```
>>> jour.append('samedi')
>>> print jour
['lundi', 'mardi', 'mercredi', 'juillet', 'jeudi', 'vendredi', 'samedi']
>>>
```

Dans la première ligne de l'exemple ci-dessus, nous avons appliqué la *méthode* **append()** à *l'objet* **jour** , avec *l'argument* **'samedi'**. Si l'on se rappelle que le mot *append* signifie "ajouter" en anglais, on peut comprendre que la méthode **append()** est une sorte de fonction qui est en quelque manière attachée ou intégrée aux objets du type "liste". L'argument que l'on utilise avec cette fonction est bien entendu l'élément que l'on veut ajouter à la fin de la liste.

Nous verrons plus loin qu'il existe ainsi toute une série de ces *méthodes* (c.à.d. des fonctions intégrées, ou plutôt "encapsulées" dans les objets de type "liste"). Notons simplement au passage que l'***on applique une méthode à un objet en reliant les deux à l'aide d'un point.*** (D'abord le nom de la variable qui référence l'objet, puis le point, puis le nom de la méthode, cette dernière toujours accompagnée d'une paire de parenthèses).

17 Il existe en fait tout un ensemble de techniques qui permettent de découper une liste en tranches, d'y insérer des groupes d'éléments, d'en enlever d'autres, etc., en utilisant une syntaxe particulière où n'interviennent que les index.
Cet ensemble de techniques (qui peuvent aussi s'appliquer aux chaînes de caractères) porte le nom générique de *slicing* (tranchage). On le met en oeuvre en plaçant plusieurs indices au lieu d'un seul entre les crochets que l'on accole au nom de la variable. Ainsi jour[1:3] désigne le sous-ensemble ['mardi', 'mercredi'].
Ces techniques un peu particulières sont décrites plus loin (voir pages 117 et suivantes).

Comme les chaînes de caractères, les listes seront approfondies plus loin dans ces notes (voir page 128). Nous en savons cependant assez pour commencer à les utiliser dans nos programmes. Veuillez par exemple analyser le petit script ci-dessous et commenter son fonctionnement :

```
jour = ['dimanche','lundi','mardi','mercredi','jeudi','vendredi','samedi']
a, b = 0, 0
while a<25:
     a = a + 1
     b = a % 7
     print a, jour[b]
```

La 5e ligne de cet exemple fait usage de l'opérateur "*modulo*", déjà rencontré précédemment et qui peut rendre de grands services en programmation. On le représente par % dans de nombreux langages (dont Python). Quelle est l'opération effectuée par cet opérateur ?

Exercices :

e 23. Soient les listes suivantes :

```
t1 = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
t2 = ['Janvier', 'Février', 'Mars', 'Avril', 'Mai', 'Juin',
      'Juillet', 'Août', 'Septembre', 'Octobre', 'Novembre', 'Décembre']
```

Écrivez un petit programme qui crée une nouvelle liste **t3**. Celle-ci devra contenir tous les éléments des deux listes en les alternant, de telle manière que chaque nom de mois soit suivi du nombre de jours correspondant : `['Janvier',31,'Février',28,'Mars',31, etc...]`.

e 24. Écrivez un programme qui affiche "proprement" tous les éléments d'une liste. Si on l'appliquait par exemple à la liste **t2** de l'exercice ci-dessus, on devrait obtenir :

```
Janvier Février Mars Avril Mai Juin Juillet Août Septembre Octobre
Novembre Décembre
```

e 25. Écrivez un programme qui recherche le plus grand élément présent dans une liste donnée. Par exemple, si on l'appliquait à la liste `[32, 5, 12, 8, 3, 75, 2, 15]`, ce programme devrait afficher :

```
le plus grand élément de cette liste a la valeur 75.
```

e 26. Écrivez un programme qui analyse un par un tous les éléments d'une liste de nombres (par exemple celle de l'exercice précédent) pour générer deux nouvelles listes. L'une contiendra seulement les nombres pairs de la liste initiale, et l'autre les nombres impairs. Par exemple, si la liste initiale est celle de l'exercice précédent, le programme devra construire une liste **pairs** qui contiendra `[32, 12, 8, 2]`, et une liste **impairs** qui contiendra `[5, 3, 75, 15]`. Astuce : pensez à utiliser l'opérateur modulo (%) déjà cité précédemment.

e 27. Écrivez un programme qui analyse un par un tous les éléments d'une liste de mots (par exemple : ['Jean', 'Maximilien', 'Brigitte', 'Sonia', 'Jean-Pierre', 'Sandra'] pour générer deux nouvelles listes. L'une contiendra les mots comportant moins de 6 caractères, l'autre les mots comportant 6 caractères ou davantage.

Chapitre 6 : Fonctions prédéfinies

L'un des concepts les plus importants en programmation est celui de ***fonction***[18]. Les fonctions permettent en effet de décomposer un programme complexe en une série de sous-programmes plus simples, lesquels peuvent à leur tour être décomposés eux-mêmes en fragments plus petits, et ainsi de suite. D'autre part, les fonctions sont réutilisables : si nous disposons d'une fonction capable de calculer une racine carrée, par exemple, nous pouvons l'utiliser un peu partout dans nos programmes sans avoir à la ré-écrire à chaque fois.

6.1 Interaction avec l'utilisateur : la fonction input()

La plupart des scripts élaborés nécessitent à un moment ou l'autre une intervention de l'utilisateur (entrée d'un paramètre, clic de souris sur un bouton, etc.). Dans un script simple en mode texte (comme ceux que nous avons créés jusqu'à présent), la méthode la plus simple consiste à employer la fonction intégrée **input().** Cette fonction provoque une interruption dans le programme courant. L'utilisateur est invité à entrer des caractères au clavier et à terminer avec <Enter>. Lorsque cette touche est enfoncée, l'exécution du programme se poursuit, et la fonction fournit en retour une valeur correspondant à ce que l'utilisateur a entré. Cette valeur peut alors être assignée à une variable quelconque.

On peut invoquer la fonction **input()** en laissant les parenthèses vides. On peut aussi y placer en argument un message explicatif destiné à l'utilisateur. Exemple :

```
print 'Veuillez entrer un nombre positif quelconque : ',
nn = input()
print 'Le carré de', nn, 'vaut', nn**2
```

Remarques importantes :

- La fonction **input()** renvoie une valeur dont le type correspondant à ce que l'utilisateur a entré. Dans notre exemple, la variable **nn** contiendra donc un entier, une chaîne de caractères, un réel, etc. suivant ce que l'utilisateur aura décidé. Si l'utilisateur souhaite entrer une chaîne de caractères, il doit l'entrer comme telle, ***c.à.d. incluse entre des apostrophes ou des guillemets***. Nous verrons plus loin qu'un bon script doit toujours vérifier si le type ainsi entré correspond bien à ce que l'on attend pour la suite du programme.
- Pour cette raison, il sera souvent préférable d'utiliser dans vos scripts la fonction similaire **raw_input()**, laquelle renvoie toujours ***une chaîne de caractères***. Vous pouvez ensuite convertir cette chaîne en nombre à l'aide de **int()** ou de **float()**. Exemple :

```
>>> a = raw_input('Entrez une donnée : ')
Entrez une donnée : 52.37
>>> type(a)
<type 'str'>
>>> b = float(a)        # conversion en valeur numérique
>>> type(b)
<type 'float'>
```

18 Sous Python, le terme de "fonction" est utilisé indifféremment pour désigner à la fois de véritables fonctions mais également des **procédures**. Nous indiquerons plus loin la distinction entre ces deux concepts proches.

6.2 Importer un module de fonctions

Vous avez déjà rencontré des fonctions ***intégrées*** au langage lui-même, comme la fonction **len()**, par exemple, qui permet de connaître la longueur d'une chaîne de caractères. Il va de soi cependant qu'il n'est pas possible d'intégrer toutes les fonctions imaginables dans le corps standard de Python, car il en existe virtuellement une infinité : vous apprendrez d'ailleurs très bientôt comment en créer vous-même de nouvelles. Les fonctions intégrées au langage sont relativement peu nombreuses : ce sont seulement celles qui sont susceptibles d'être utilisées très fréquemment. Les autres sont regroupées dans des fichiers séparés que l'on appelle des ***modules***.

Les modules sont donc des fichiers qui regroupent des ensembles de fonctions. Vous verrez plus loin comme il est commode de découper un programme important en plusieurs fichiers de taille modeste pour en faciliter la maintenance. Une application Python typique sera alors constituée d'un programme principal accompagné de un ou plusieurs modules contenant chacun les définitions d'un certain nombre de fonctions accessoires.

Il existe un grand nombre de modules pré-programmés qui sont fournis d'office avec Python. Vous pouvez en trouver d'autres chez divers fournisseurs. Souvent on essaie de regrouper dans un même module des ensembles de fonctions apparentées que l'on appelle des ***bibliothèques***.

Le module **math**, par exemple, contient les définitions de nombreuses fonctions mathématiques telles que *sinus, cosinus, tangente, racine carrée*, etc. Pour pouvoir utiliser ces fonctions, il vous suffit d'incorporer la ligne suivante au début de votre script :

```
from math import *
```

Cette ligne indique à Python qu'il lui faut inclure dans le programme courant ***toutes*** les fonctions (c'est là la signification du symbole *) du module *math*, lequel contient une bibliothèque de fonctions mathématiques pré-programmées.

Dans le corps du script lui-même, vous écrirez par exemple :

`racine = sqrt(nombre)` pour assigner à la variable *racine* la racine carrée de *nombre,*
`sinusx = sin(angle)` pour assigner à la variable *sinusx* le sinus de *angle* (en radians !), etc.

Exemple :

```
# Démo : utilisation des fonctions du module <math>

from math import *

nombre = 121
angle = pi/6  # soit 30° (la bibliothèque math inclut aussi la définition de pi)

print 'racine carrée de', nombre, '=', sqrt(nombre)
print 'sinus de', angle, 'radians', '=', sin(angle)
```

L'exécution de ce script provoque l'affichage suivant :

```
racine carrée de 121 = 11.0
sinus de 0.523598775598 radians = 0.5
```

Ce court exemple illustre déjà fort bien quelques caractéristiques importantes des fonctions :

- une fonction apparaît sous la forme d'***un nom quelconque associé à des parenthèses***
  exemple : **`sqrt()`**
- dans les parenthèses, on ***transmet*** à la fonction un ou plusieurs ***arguments***
  exemple : **`sqrt(121)`**
- la fonction fournit une ***valeur de retour*** (on dira aussi qu'elle "renvoie" ou "retourne" une valeur)
  exemple : **`11.0`**

Nous allons développer tout ceci dans les pages suivantes. Veuillez noter au passage que les fonctions mathématiques utilisées ici ne représentent qu'un tout premier exemple. Un simple coup d'oeil dans la documentation des bibliothèques Python vous permettra de constater que de très nombreuses fonctions sont d'ores et déjà disponibles pour réaliser une multitude de tâches, y compris des algorithmes mathématiques très complexes (Python est couramment utilisé dans les universités pour la résolution de problèmes scientifiques de haut niveau). Il est donc hors de question de fournir ici une liste détaillée. Une telle liste est aisément accessible dans le système d'aide de Python :

Documentation HTML → Python documentation → Modules index → math

Au chapitre suivant, nous apprendrons comment créer nous-mêmes de nouvelles fonctions.

Exercices :

(Note : Dans tous ces exercices, utilisez la fonction input() pour l'entrée des données)

e 28. Écrivez un programme qui convertisse en mètres par seconde et en km/h une vitesse fournie par l'utilisateur en miles/heure. (*Rappel : 1 mile = 1609 mètres*)

e 29. Écrivez un programme qui calcule le périmètre et l'aire d'un triangle quelconque dont l'utilisateur fournit les 3 côtés.
(*Rappel : l'aire d'un triangle quelconque se calcule à l'aide de la formule :*

$$S=\sqrt{d\cdot(d-a)\cdot(d-b)\cdot(d-c)}$$

dans laquelle d désigne la longueur du demi-périmètre, et a, b, c celles des trois côtés).

e 30. Écrivez un programme qui calcule la période d'un pendule simple de longueur donnée.
La formule qui permet de calculer la période d'un pendule simple est $T=2\pi\sqrt{\frac{l}{g}}$,
l représentant la longueur du pendule et *g* la valeur de l'accélération de la pesanteur au lieu d'expérience.

e 31. Écrivez un programme qui permette d'encoder des valeurs dans une liste. Ce programme devrait fonctionner en boucle, l'utilisateur étant invité à entrer sans cesse de nouvelles valeurs, jusqu'à ce qu'il décide de terminer en frappant <enter> en guise d'entrée. Le programme se terminerait alors par l'affichage de la liste. Exemple de fonctionnement :

```
Veuillez entrer une valeur : 25
Veuillez entrer une valeur : 18
Veuillez entrer une valeur : 6284
Veuillez entrer une valeur :
[25, 18, 6284]
```

6.3 Un peu de détente avec le module turtle

Comme nous venons de le voir, l'une des grandes qualités de Python est qu'il est extrêmement facile de lui ajouter de nombreuses fonctionnalités par importation de divers ***modules***.

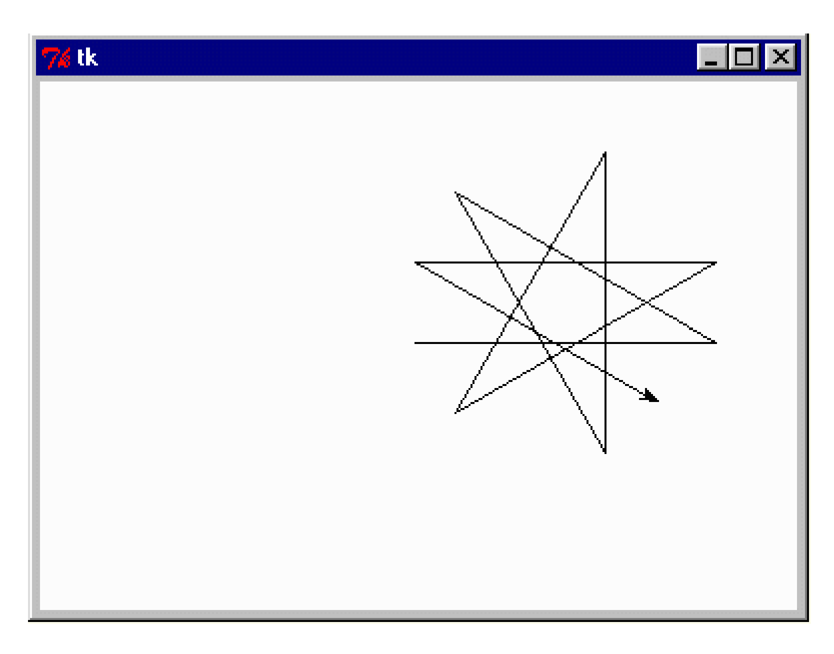
Pour illustrer cela, et nous amuser un peu avec d'autres objets que des nombres, nous allons explorer un module Python qui permet de réaliser des "graphiques tortue", c.à.d. des dessins géométriques correspondant à la piste laissée derrière elle par une petite "tortue" virtuelle dont nous contrôlons les déplacements sur l'écran de l'ordinateur à l'aide d'instructions simples.

Activer cette tortue est un vrai jeu d'enfant. Plutôt que de vous donner de longues explications, nous vous invitons à essayer tout de suite :

```
>>> from turtle import *
>>> forward(120)
>>> left(90)
>>> color('red')
>>> forward(80)
```

L'exercice est évidemment plus riche si l'on utilise des boucles :

```
>>> reset()
>>> a = 0
>>> while a <12:
        a = a +1
        forward(150)
        left(150)
```



Attention cependant : avant de lancer un tel script, assurez-vous toujours qu'il ne comporte pas de boucle sans fin (voir page 36), car si c'est le cas vous risquez de ne plus pouvoir reprendre le contrôle des opérations (en particulier sous Window$).

Amusez-vous à écrire des scripts qui réalisent des dessins suivant un modèle imposé à l'avance. Les principales fonctions mises à votre disposition dans le module *turtle* sont les suivantes :

reset()	On efface tout et on recommence
goto(x, y)	Aller à l'endroit de coordonnées x, y
forward(distance)	Avancer d'une distance donnée
backward(distance)	Reculer
up()	Relever le crayon (pour pouvoir avancer sans dessiner)
down()	Abaisser le crayon(pour recommencer à dessiner)
color(couleur)	<couleur> peut être une chaîne prédéfinie ('red', 'blue', 'green', etc.)
left(angle)	Tourner à gauche d'un angle donné (exprimé en degrés)
right(angle)	Tourner à droite
width(épaisseur)	Choisir l'épaisseur du tracé
fill(1)	Remplir un contour fermé à l'aide de la couleur sélectionnée
write(texte)	<texte> doit être une chaîne de caractères délimitée avec des " ou des '

6.4 Véracité/fausseté d'une expression

Lorsqu'un programme contient des instructions telles que **while** ou **if**, l'ordinateur qui exécute ce programme doit évaluer la véracité d'une condition, c.à.d. déterminer si une expression est vraie ou fausse. Par exemple, une boucle initiée par **while c<20:** s'exécutera aussi longtemps que la condition **c<20** restera ***vraie***.

Mais comment un ordinateur peut-il déterminer si quelque chose est vrai ou faux ?

En fait - et vous le savez déjà - un ordinateur ne manipule strictement que des nombres. Tout ce qu'un ordinateur doit traiter doit d'abord toujours être converti en valeur numérique. Cela s'applique aussi à la notion de vrai/faux. En Python, tout comme en *C*, en *Basic* et en de nombreux autres langages de programmation, on considère que toute valeur numérique autre que zéro est "vraie". Seule la valeur zéro est "fausse". Exemple :

```
a = input('Entrez une valeur quelconque')
if a:
    print "vrai"
else:
    print "faux"
```

Le petit script ci-dessus n'affiche "faux" que si vous entrez la valeur 0. Pour toute autre valeur numérique, vous obtiendrez "vrai".

Si vous entrez une chaîne de caractères ou une liste, vous obtiendrez encore "vrai". Seules les chaînes ou les listes ***vides*** seront considérées comme "fausses".

Tout ce qui précède signifie donc qu'une expression à évaluer, telle par exemple la condition $\mathbf{a > 5}$, est d'abord convertie par l'ordinateur en une valeur numérique. (Généralement 1 si l'expression est vraie, et zéro si l'expression est fausse). Exemple :

```
a = input('entrez une valeur numérique : ')
b = (a < 5)
print 'la valeur de b est', b, ':'
if b:
    print "la condition b est vraie"
else:
    print "la condition b est fausse"
```

Le script ci-dessus vous retourne une valeur **b = 1** (condition vraie) si vous avez entré un nombre plus petit que 5.

Ces explications ne sont qu'une première information à propos d'un système de représentation des opérations logiques que l'on appelle ***algèbre de Boole***. Vous apprendrez plus loin que l'on peut appliquer aux nombres binaires des opérateurs tels que **and**, **or**, **not**, etc. qui permettent d'effectuer à l'aide de ces nombres des traitements logiques complexes.

6.5 Révision

Dans ce qui suit, nous n'allons pas apprendre de nouveaux concepts mais simplement utiliser tout ce que nous connaissons déjà pour réaliser de vrais petits programmes.

6.5.1 Contrôle du flux - Utilisation d'une liste simple

Commençons par un petit retour sur les branchements conditionnels (il s'agit peut-être là du groupe d'instructions le plus important dans n'importe quel langage !) :

```
# Utilisation d'une liste et de branchements conditionnels

print "Ce script recherche le plus grand de trois nombres"
print 'Veuillez entrer trois nombres séparés par des virgules : '
# Note : la fonction list() convertit en liste la séquence de données qu'on
# lui fournit en argument. L'instruction ci-dessous convertira donc les
# données fournies par l'utilisateur en une liste  nn :
nn = list(input())
max, index = nn[0], 'premier'
if nn[1] > max:              # ne pas omettre le double point !
    max = nn[1]
    index = 'second'
if nn[2] > max:
    max = nn[2]
    index = 'troisième'
print "Le plus grand de ces nombres est", max
print "Ce nombre est le", index, "de votre liste."
```

Note : Dans cet exercice, vous retrouvez à nouveau le concept de "bloc d'instructions", déjà abondamment commenté aux chapitres 3 et 4, et que vous devez absolument assimiler. Pour rappel, les blocs d'instructions sont délimités ***par l'indentation***. Après la première instruction **if**, par exemple, il y a deux lignes indentées définissant un bloc d'instructions. Ces instructions ne seront exécutées que si la condition **nn[1] > max** est vraie.

La ligne suivante, par contre (celle qui contient la deuxième instruction **if**) n'est pas indentée. Cette ligne se situe donc au même niveau que celles qui définissent le corps principal du programme. L'instruction contenue dans cette ligne est donc toujours exécutée, alors que les deux suivantes (qui constituent encore un autre bloc) ne sont exécutées que si la condition **nn[2] > max** est vraie.

En suivant la même logique, on voit que les instructions des deux dernières lignes font partie du bloc principal et sont donc toujours exécutées.

6.5.2 *Boucle while - Instructions imbriquées*

Continuons dans cette voie en imbriquant d'autres structures :

```
# Instructions composées <while> - <if> - <elif> - <else>                #  1

print 'Choisissez un nombre de 1 à 3 (ou zéro pour terminer) ',          #  3
a = input()                                                              #  4
while a != 0:            # l'opérateur != signifie "différent de"        #  5
    if a == 1:                                                           #  6
        print "Vous avez choisi un :"                                    #  7
        print "le premier, l'unique, l'unité ..."                        #  8
    elif a == 2:                                                         #  9
        print "Vous préférez le deux :"                                  # 10
        print "la paire, le couple, le duo ..."                          # 11
    elif a == 3:                                                         # 12
        print "Vous optez pour le plus grand des trois :"                # 13
        print "le trio, la trinité, le triplet ..."                      # 14
    else :                                                               # 15
        print "Un nombre entre UN et TROIS, s.v.p."                      # 16
    print 'Choisissez un nombre de 1 à 3 (ou zéro pour terminer) ',      # 17
    a = input()                                                          # 18
print "Vous avez entré zéro :"                                           # 19
print "L'exercice est donc terminé."                                     # 20
```

Nous retrouvons ici une boucle **while**, associée à un groupe d'instructions **if, elif** et **else**. Notez bien cette fois encore comment la structure logique du programme est créée à l'aide des indentations (... et n'oubliez pas le caractère "**:**" à la fin de chaque ligne d'en-tête !)

L'instruction **while** est utilisée ici pour relancer le questionnement après chaque réponse de l'utilisateur (du moins jusqu'à ce que celui-ci décide de "sortir" en entrant une valeur nulle : rappelons à ce sujet que l'opérateur de comparaison **!=** signifie "est différent de"). Dans le corps de la boucle, nous trouvons le groupe d'instructions **if, elif** et **else** (de la ligne 6 à la ligne 16), qui aiguille le flux du programme vers les différentes réponses, ensuite une instruction **print** et une instruction **input()** (lignes 17 & 18) qui seront exécutées dans tous les cas de figure : notez bien leur niveau d'indentation, qui est le même que celui du bloc **if, elif** et **else**, Après ces instructions, le programme boucle et l'exécution reprend à l'instruction **while** (ligne 5). Les deux dernières instructions **print** (lignes 19 & 20) ne sont exécutées qu'à la sortie de la boucle.

6.5.3 Exercices

e 32. Que fait le programme ci-dessous, dans les quatre cas où l'on aurait défini au préalable que la variable **a** vaut 1, 2, 3 ou 15 ?

```
if a !=2:
    print 'perdu'
elif a ==3:
    print 'un instant, s.v.p.'
else :
    print 'gagné'
```

e 33. Que font ces programmes ?

a)
```
a = 5
b = 2
if (a==5) & (b<2):
     print '"&" signifie "et"; on peut aussi utiliser le mot "and"'
```

b)
```
a, b = 2, 4
if (a==4) or (b!=4):
     print 'gagné'
elif (a==4) or (b==4):
     print 'presque gagné'
```

c)
```
a = 1
if not a:
     print 'gagné'
elif a:
     print 'perdu'
```

Reprendre le programme c) avec a = 0 au lieu de a = 1. Que se passe-t-il ? Conclure !

e 34. Écrire un programme qui, étant données deux bornes entières a et b, additionne les nombres multiples de 3 **et** de 5 compris entre ces bornes.
Prendre par exemple a = 0, b = 32 → le résultat devrait être alors 0 + 15 + 30 = 45.
Modifier légèrement ce programme pour qu'il additionne les nombres multiples de 3 **ou** de 5 compris entre les bornes a et b. Avec les bornes 0 et 32, le résultat devrait donc être : 0 + 3 + 5 + 6 + 9 + 10 + 12 + 15 + 18 + 20 + 21 + 24 + 25 + 27 + 30 = 225.

e 35. Déterminer si une année (dont le millésime est introduit par l'utilisateur) est bissextile ou non. (Une année A est bissextile si A est divisible par 4. Elle ne l'est cependant pas si A est un multiple de 100, à moins que A ne soit multiple de 400).

e 36. Demander à l'utilisateur son nom et son sexe (M ou F). En fonction de ces données, afficher "Cher Monsieur" ou "Chère Mademoiselle" suivi du nom de l'élève.

e 37. Demander à l'utilisateur d'entrer trois longueurs a, b, c. A l'aide de ces trois longueurs, déterminer s'il est possible de construire un triangle. Déterminer ensuite si ce triangle est rectangle, isocèle, équilatéral ou quelconque. Attention : un triangle rectangle peut être isocèle.

e 38. Demander à l'utilisateur qu'il entre un nombre. Afficher ensuite : soit la racine carrée de ce nombre, soit un message indiquant que la racine carrée de ce nombre ne peut être calculée.

e 39. Convertir une note scolaire N quelconque, entrée par l'utilisateur sous forme de points (par exemple 27 sur 85), en une note standardisée suivant le code suivant :

Note	Appréciation
N >= 80 %	A
80 % > N >= 60 %	B
60 % > N >= 50 %	C
50 % > N >= 40 %	D
N < 40 %	E

e 40. Soit la liste suivante :
['Jean-Michel', 'Marc', 'Vanessa', 'Anne', Maximilien', 'Alexandre-Benoît', 'Louise']
Ecrivez un script qui affiche chacun de ces noms avec le nombre de caractères correspondant.

e 41. Écrire une boucle de programme qui demande à l'utilisateur d'entrer des notes d'élèves. La boucle se terminera seulement si l'utilisateur entre une valeur négative. Avec les notes ainsi entrées, construire progressivement une liste. Après chaque entrée d'une nouvelle note (et donc à chaque itération de la boucle), afficher le nombre de notes entrées, la note la plus élevée, la note la plus basse, la moyenne de toutes les notes.

e 42. Ecrivez un script qui affiche la valeur de la force de gravitation s'exerçant entre deux masses de 10000 kg , pour des distances qui augmentent suivant une progression géométrique de raison 2, à partir de 5 cm (0,05 mètre).

La force de gravitation est régie par la formule $F = 6{,}67\,10^{-11} \cdot \frac{m \cdot m'}{d^2}$

Exemple d'affichage :
```
d = .05 m :  la force vaut  2.668 N
d = .1 m  :  la force vaut  0.667 N
d = .2 m  :  la force vant  0.167 N
d = .4 m  :  la force vaut  0.0417 N
etc.
```

Chapitre 7 : Fonctions originales

La programmation est l'art d'apprendre à un ordinateur à accomplir des tâches qu'il n'était pas capable de réaliser auparavant. L'une des méthodes les plus intéressantes pour y arriver consiste à ajouter de nouvelles instructions au langage de programmation que vous utilisez, sous la forme de fonctions originales.

7.1 Définir une fonction

Les scripts que vous avez écrits jusqu'à présent étaient à chaque fois très courts, car leur objectif était seulement de vous faire assimiler les premiers éléments du langage. Lorsque vous commencerez à développer de véritables projets, vous serez confrontés à des problèmes souvent fort complexes, et les lignes de programme vont commencer à s'accumuler...

L'approche efficace d'un problème complexe consiste souvent à le décomposer en plusieurs sous-problèmes plus simples qui seront étudiés séparément (Ces sous-problèmes peuvent éventuellement être eux-mêmes décomposés à leur tour, et ainsi de suite). Or il est important que cette décomposition soit représentée fidèlement dans les algorithmes pour que ceux-ci restent clairs.

D'autre part, il arrivera souvent qu'une même séquence d'instructions doive être utilisée à plusieurs reprises dans un programme, et on souhaitera bien évidemment ne pas avoir à la reproduire systématiquement.

Les ***fonctions***[19] et les ***classes d'objets*** sont différentes structures de sous-programmes qui ont été imaginées par les concepteurs des langages de haut niveau afin de résoudre les difficultés évoquées ci-dessus. Nous allons commencer par décrire ici la définition de fonctions sous Python. Les objets et les classes seront examinés plus loin.

Nous avons déjà rencontré diverses fonctions pré-programmées. Voyons à présent comment en définir nous-mêmes de nouvelles.

La syntaxe Python pour la définition d'une fonction est la suivante :

```
def nomDeLaFonction(liste de paramètres):
    ...
    bloc d'instructions
    ...
```

- Vous pouvez choisir n'importe quel nom pour la fonction que vous créez, à l'exception des mots réservés du langage[20], et à la condition de n'utiliser aucun caractère spécial ou accentué (le caractère souligné "_" est permis). Comme c'est le cas pour les noms de variables, il vous est conseillé d'utiliser surtout des lettres minuscules, notamment au début du nom (les noms commençant par une majuscule seront réservés aux ***classes*** que nous étudierons plus loin).

19 Il existe aussi dans d'autres langages des **routines** (parfois appelés sous-programmes) et des **procédures**. Python utilise le même terme de **fonction**, pour désigner à la fois les fonctions au sens strict (qui fournissent une valeur en retour), et les procédures (qui n'en fournissent pas).

20 La liste complète des mots réservés Python se trouve page 22.

- Comme les instructions **if** et **while** que vous connaissez déjà, l'instruction **def** est une *instruction composée*. La ligne contenant cette instruction se termine obligatoirement par un double point, lequel introduit un bloc d'instructions que vous ne devez pas oublier d'indenter.
- La ***liste de paramètres*** spécifie quelles informations il faudra fournir en guise d'arguments lorsque l'on voudra utiliser cette fonction (Les parenthèses peuvent parfaitement rester vides si la fonction ne nécessite pas d'arguments).
- Une fonction s'utilise pratiquement comme une instruction quelconque. Dans le corps d'un programme, un ***appel de fonction*** est constitué du nom de la fonction suivi de parenthèses.
  Si c'est nécessaire, on place dans ces parenthèses le ou les arguments que l'on souhaite transmettre à la fonction. Il faudra en principe fournir un argument pour chacun des paramètres spécifiés dans la définition de la fonction, encore qu'il soit possible de définir pour ces paramètres des valeurs par défaut (voir plus loin).

7.1.1 Fonction simple sans paramètres

Pour notre première approche concrète des fonctions, nous allons travailler à nouveau en mode interactif. Le mode interactif de Python est en effet idéal pour effectuer des petits tests comme ceux qui suivent. C'est une facilité que n'offrent pas dans tous les langages de programmation !

```
>>> def table7():
...     n = 1
...     while n <11 :
...         print n * 7,
...         n = n+1
...
```

En entrant ces quelques lignes, nous avons défini une fonction très simple qui calcule et affiche les 10 premiers termes de la table de multiplication par 7. Notez bien les parenthèses[21], le double point, et l'indentation du bloc d'instructions qui suit la ligne d'en-tête (c'est ce bloc d'instructions qui constitue le corps de la fonction proprement dite).

Pour utiliser la fonction que nous venons de définir, il suffit de l'appeler par son nom. Ainsi :

```
>>> table7()
```

provoque l'affichage de :

```
7 14 21 28 35 42 49 56 63 70
```

21 Un nom de fonction doit toujours être accompagné de parenthèses, même si la fonction n'utilise aucun paramètre.
Il en résulte une convention d'écriture qui stipule que dans un texte quelconque traitant de programmation d'ordinateur, un nom de fonction soit toujours accompagné d'une paire de parenthèses vides.
Nous respecterons cette convention dans la suite de ce texte.

Nous pouvons maintenant réutiliser cette fonction à plusieurs reprises, autant de fois que nous le souhaitons. Nous pouvons également l'incorporer dans la définition d'une autre fonction, comme dans l'exemple ci-dessous :

```
>>> def table7triple():
...     print 'La table par 7 en triple exemplaire :'
...     table7()
...     table7()
...     table7()
...
```

Utilisons cette nouvelle fonction, en entrant la commande :

```
>>> table7triple()
```

l'affichage résultant devrait être :

```
La table par 7 en triple exemplaire :
7 14 21 28 35 42 49 56 63 70
7 14 21 28 35 42 49 56 63 70
7 14 21 28 35 42 49 56 63 70
```

Une première fonction peut donc appeler une deuxième fonction, qui elle-même en appelle une troisième, etc. Au stade où nous sommes, vous ne voyez peut-être pas encore très bien l'utilité de tout cela, mais vous pouvez déjà noter deux propriétés intéressantes :

- Créer une nouvelle fonction vous offre l'opportunité de donner un nom à tout un ensemble d'instructions. De cette manière, vous pouvez simplifier le corps principal d'un programme, en dissimulant un algorithme secondaire complexe sous une commande unique, à laquelle vous pouvez donner un nom très explicite, en français si vous voulez.
- Créer une nouvelle fonction peut servir à raccourcir un programme, par élimination des portions de code qui se répètent. Par exemple, si vous devez afficher la table par 7 plusieurs fois dans un même programme, vous n'avez pas à réécrire chaque fois l'algorithme qui accomplit ce travail.

Une fonction est donc en quelque sorte une nouvelle instruction personnalisée, que vous ajoutez vous-même librement à votre langage de programmation.

7.1.2 Fonction avec paramètre

Dans nos derniers exemples, nous avons défini et utilisé une fonction qui affiche les termes de la table par 7. Supposons à présent que nous voulions faire de même avec la table par 9. Nous pouvons bien entendu réécrire entièrement une nouvelle fonction pour cela. Mais si nous nous intéressons plus tard à la table par 13, il nous faudra encore recommencer. Ne serait-il donc pas plus intéressant de définir une fonction qui soit capable d'afficher n'importe quelle table, à la demande ?

Lorsque nous appellerons cette fonction, nous devrons bien évidemment pouvoir lui indiquer quelle table nous souhaitons afficher. Cette information que nous voulons transmettre à la fonction au moment même où nous l'appelons s'appelle un ***argument***. Nous avons déjà rencontré à plusieurs reprises des fonctions intégrées qui utilisent des arguments. La fonction **sin(a)**, par exemple, calcule le sinus de l'angle **a**. La fonction **sin()** utilise donc la valeur numérique de **a** comme argument pour effectuer son travail.

Dans la définition d'une telle fonction, il faut prévoir une variable particulière pour recevoir l'argument transmis. Cette variable particulière s'appelle un ***paramètre.*** On lui choisit un nom en respectant les mêmes règles de syntaxe que d'habitude (pas de lettres accentuées, etc.), et on place ce nom entre les parenthèses qui accompagnent la définition de la fonction.

Voici ce que cela donne dans le cas qui nous intéresse :

```
>>> def table(base):
...     n = 1
...     while n <11 :
...        print n * base,
...        n = n +1
```

La fonction **table()** telle que définie ci-dessus utilise le paramètre **base** pour calculer les dix premiers termes de la table de multiplication correspondante.

Pour tester cette nouvelle fonction, il nous suffit de l'appeler avec un argument. Exemples :

```
>>> table(13)
13 26 39 52 65 78 91 104 117 130

>>> table(9)
9 18 27 36 45 54 63 72 81 90
```

Dans ces exemples, la valeur que nous indiquons entre parenthèses lors de l'appel de la fonction (et qui est donc un argument) est automatiquement affectée au paramètre **base**. Dans le corps de la fonction, **base** joue le même rôle que n'importe quelle autre variable. Lorsque nous entrons la commande **table(9)**, nous signifions à la machine que nous voulons exécuter la fonction **table()** en affectant la valeur **9** à la variable **base**.

7.1.3 Utilisation d'une variable comme argument

Dans les 2 exemples qui précèdent, l'argument que nous avons utilisé en appelant la fonction table() était à chaque fois une constante (la valeur 13, puis la valeur 9). Cela n'est nullement obligatoire. ***L'argument que nous utilisons dans l'appel d'une fonction peut être une variable*** lui aussi, comme dans l'exemple ci-dessous. Analysez bien cet exemple, essayez-le concrètement, et décrivez le mieux possible dans votre cahier d'exercices ce que vous obtenez, en expliquant avec vos propres mots ce qui se passe. Cet exemple devrait vous donner un premier aperçu de l'utilité des fonctions pour accomplir simplement des tâches complexes :

```
>>> a = 1
>>> while a <20:
...     table(a)
...     a = a +1
...2
```

Remarque importante :

Dans l'exemple ci-dessus, l'argument que nous passons à la fonction **table()** est le contenu de la variable **a** . A l'intérieur de la fonction, cet argument est affecté au paramètre **base**, qui est une tout autre variable. Notez donc bien dès à présent que :

Le nom d'une variable que nous passons comme argument n'a rien à voir avec le nom du paramètre correspondant dans la fonction.

Ces noms peuvent être identiques si vous le voulez, mais vous devez bien comprendre qu'ils ne désignent pas la même chose (en dépit du fait qu'ils puissent contenir une valeur identique).

Exercice :

e 43. Importez le module ***turtle*** pour pouvoir effectuer des dessins simples.

Vous allez dessiner une série de triangles équilatéraux de différentes couleurs.
Pour ce faire, définissez d'abord une fonction **triangle()** capable de dessiner un triangle d'une couleur bien déterminée (ce qui signifie donc que la définition de votre fonction doit comporter un paramètre pour recevoir le nom de cette couleur)

Utilisez ensuite cette fonction pour reproduire ce même triangle en différents endroits, en changeant de couleur à chaque fois.

7.1.4 Fonction avec plusieurs paramètres

La fonction **table()** est certainement intéressante, mais elle n'affiche toujours que les dix premiers termes de la table de multiplication, alors que nous pourrions souhaiter qu'elle en affiche d'autres. Qu'à cela ne tienne. Nous allons l'améliorer en lui ajoutant des paramètres supplémentaires, dans une nouvelle version que nous appellerons cette fois **tableMulti()** :

```
>>> def tableMulti(base, debut, fin):
...     print 'Fragment de la table de multiplication par', base, ':'
...     n = debut
...     while n <= fin :
...        print n, 'x', base, '=', n * base
...        n = n +1
```

Cette nouvelle fonction utilise donc trois paramètres : la base de la table comme dans l'exemple précédent, l'indice du premier terme à afficher, l'indice du dernier terme à afficher.

Essayons cette fonction en entrant par exemple :

```
>>> tableMulti(8, 13, 17)
```

ce qui devrait provoquer l'affichage de :

```
Fragment de la table de multiplication par 8 :
13 x 8 = 104
14 x 8 = 112
15 x 8 = 120
16 x 8 = 128
17 x 8 = 136
```

Notes :

- Pour définir une fonction avec plusieurs paramètres, il suffit d'inclure ceux-ci entre les parenthèses qui suivent le nom de la fonction, en les séparant à l'aide de virgules.
- Lors de l'appel de la fonction, les arguments utilisés doivent être fournis ***dans le même ordre*** que celui des paramètres correspondants (en les séparant eux aussi à l'aide de virgules). Le premier argument sera affecté au premier paramètre, le second argument sera affecté au second paramètre, et ainsi de suite.
- A titre d'exercice, essayez la séquence d'instructions suivantes et décrivez dans votre cahier d'exercices le résultat obtenu :

```
>>> t, d, f = 11, 5, 10
>>> while t<21:
...     tableMulti(t,d,f)
...     t, d, f = t +1, d +3, f +5
...
```

7.2 Variables locales, variables globales

Lorsque nous définissons des variables à l'intérieur du corps d'une fonction, ces variables ne sont accessibles qu'à la fonction elle-même. On dit que ces variables sont des ***variables locales*** à la fonction. C'est par exemple le cas des variables **base**, **debut**, **fin** et **n** dans l'exercice précédent.

Chaque fois que la fonction **tableMulti()** est appelée, Python réserve pour elle (dans la mémoire de l'ordinateur) ***un nouvel espace de noms***[22]. Les contenus des variables **base**, **debut**, **fin** et **n** sont stockés dans cet espace de noms qui est inaccessible depuis l'extérieur de la fonction. Ainsi par exemple, si nous essayons d'afficher le contenu de la variable **base** juste après avoir effectué l'exercice ci-dessus, nous obtenons un message d'erreur :

```
>>> print base
Traceback (innermost last):
  File "<pyshell#8>", line 1, in ?
    print base
NameError: base
```

La machine nous signale clairement que le symbole **base** lui est inconnu, alors qu'il était correctement imprimé par la fonction **tableMulti()** elle-même. L'espace de noms qui contient le symbole **base** est strictement réservé au fonctionnement interne de **tableMulti()**, et il est automatiquement détruit dès que la fonction a terminé son travail.

Les variables définies ***à l'extérieur*** d'une fonction sont des ***variables globales***. Leur contenu est "visible" de l'intérieur d'une fonction, mais la fonction ne peut pas le modifier. Exemple :

```
>>> def mask():
...     p = 20
...     print p, q
...
>>> p, q = 15, 38
>>> mask()
20 38
>>> print p, q
15 38
```

Analysons attentivement cet exemple :

Nous commençons par définir une fonction très simple (qui n'utilise d'ailleurs aucun paramètre). A l'intérieur de cette fonction, une variable **p** est définie, avec 20 comme valeur initiale. Cette variable **p** qui est définie à l'intérieur d'une fonction sera donc une ***variable locale***.

Une fois terminée la définition de la fonction, nous revenons au niveau principal pour y définir les deux variables **p** et **q** auxquelles nous attribuons les contenus 15 et 38. Ces deux variables définies au niveau principal seront donc des ***variables globales***.

Ainsi le même nom de variable **p** a été utilisé ici à deux reprises, ***pour définir deux variables différentes*** : l'une est globale et l'autre est locale. On peut constater dans la suite de l'exercice que ces deux variables sont bel et bien des variables distinctes, indépendantes, obéissant à une règle de priorité qui veut qu'à l'intérieur d'une fonction (où elles pourraient entrer en compétition), ce sont les variables définies localement qui ont la priorité.

22 Ce concept d'*espace de noms* sera approfondi progressivement. Vous apprendrez également plus loin que les fonctions sont en fait des *objets* dont on crée à chaque fois une nouvelle *instance* lorsqu'on les appelle.

On constate en effet que lorsque la fonction **mask()** est lancée, la variable globale **q** y est accessible, puisqu'elle est imprimée correctement. Pour **p**, par contre, c'est la valeur attribuée localement qui est affichée.

On pourrait croire d'abord que la fonction **mask()** a simplement modifié le contenu de la variable globale **p** (puisqu'elle est accessible). Les lignes suivantes démontrent qu'il n'en est rien : en dehors de la fonction **mask()**, la variable globale **p** conserve sa valeur initiale.

Tout ceci peut vous paraître compliqué au premier abord. Vous comprendrez cependant très vite combien il est utile que des variables soient ainsi définies comme étant locales, c.à.d. en quelque sorte confinées à l'intérieur d'une fonction. Cela signifie en effet que vous pourrez toujours utiliser quantités de fonctions sans vous préoccuper le moins du monde des noms de variables qui y sont utilisées : ces variables ne pourront en effet jamais interférer avec celles que vous aurez vous-même définies par ailleurs.

Cet état de choses peut toutefois être modifié si vous le souhaitez. Il peut se faire par exemple que vous ayez à définir une fonction qui soit capable de modifier une variable globale. Pour atteindre ce résultat, il vous suffira d'utiliser l'instruction **global.** Cette instruction permet d'indiquer - à l'intérieur de la définition d'une fonction - quelles sont les variables à traiter globalement.

Dans l'exemple ci-dessous, la variable **a** utilisée à l'intérieur de la fonction **monter()** est non seulement accessible, mais également modifiable, parce qu'elle est signalée explicitement comme étant une variable qu'il faut traiter globalement. Par comparaison, essayez le même exercice en supprimant l'instruction **global** : la variable **a** n'est plus incrémentée à chaque appel de la fonction.

```
>>> def monter():
...  global a
...  a = a+1
...  print a
...
>>> a = 15
>>> monter()
16
>>> monter()
17
>>>
```

7.3 "Vraies" fonctions et procédures

Pour les puristes, les fonctions que nous avons décrites jusqu'à présent ne sont pas tout à fait des fonctions au sens strict, mais plus exactement des procédures[23]. Une "vraie" fonction (au sens strict) doit en effet ***renvoyer*** (ou "retourner") une valeur lorsqu'elle se termine. Une "vraie" fonction peut s'utiliser à la droite du signe *égale* dans des expressions telles que `y = sin(a)`. On comprend aisément que dans cette expression, la fonction **sin()** retourne une valeur (le sinus de l'argument **a**) qui est directement affectée à la variable y.

Commençons par un exemple extrêmement simple :

```
>>> def cube(w):
...     return w*w*w
...
```

L'instruction **return** définit ce que doit être la valeur renvoyée par la fonction. En l'occurrence, il s'agit du cube de l'argument qui a été transmis lors de l'appel de la fonction. Exemple :

```
>>> b = cube(9)
>>> print b
729
```

A titre d'exemple un peu plus élaboré, nous allons maintenant modifier quelque peu la fonction **table()** sur laquelle nous avons déjà pas mal travaillé, afin qu'elle retourne elle aussi une valeur. Cette valeur sera en l'occurrence une liste (la liste des dix premiers termes de la table de multiplication choisie). Voilà donc une occasion de reparler des listes. Nous en profiterons pour apprendre dans la foulée encore un nouveau concept :

```
>>> def table(base):
...     result = []              # result est d'abord une liste vide
...     n = 1
...     while n < 11:
...        b = n * base
...        result.append(b)      # ajout d'un terme à la liste
...        n = n +1              # (voir explications ci-dessous)
...     return result
...
2
```

Pour tester cette fonction, nous pouvons entrer par exemple :

```
>>> ta9 = table(9)
```

Ainsi nous affectons à la variable ta9 les dix premiers termes de la table de multiplication par 9, sous la forme d'une liste :

```
>>> print ta9
[9, 18, 27, 36, 45, 54, 63, 72, 81, 90]
>>> print ta9[0]
9
>>> print ta9[3]
36
>>> print ta9[2:5]
[27, 36, 45]
>>>
```

(Rappel : le premier élément d'une liste correspond à l'indice 0)

23 Dans certains langages de programmation, les fonctions et les procédures sont définies à l'aide d'instructions différentes. Python utilise la même instruction **def** pour définir les unes et les autres.

Notes:

- Comme nous l'avons vu dans l'exemple précédent, l'instruction **return** définit ce que doit être la valeur "retournée" ou "renvoyée" par la fonction. En l'occurrence, il s'agit ici du contenu de la variable **result**, c.à.d. la liste des nombres générés par la fonction[24].

- L'instruction **result.append(b)** est notre second exemple de l'utilisation d'un concept important sur lequel nous reviendrons encore abondamment par la suite : *dans cette instruction, nous appliquons la **méthode** append() à l'**objet** result.*

  Nous préciserons petit à petit ce qu'il faut entendre par ***objet*** en programmation. Pour l'instant, admettons simplement que ce terme très général s'applique notamment aux listes de Python.
  Une ***méthode*** n'est en fait rien d'autre qu'une fonction (que vous pouvez d'ailleurs reconnaître comme telle à la présence des parenthèses), mais une fonction qui est associée à un objet.
  Elle fait partie de la définition de cet objet, ou plus précisément de la ***classe*** particulière à laquelle cet objet appartient (nous étudierons ce concept de *classe* plus tard).

  Mettre en œuvre une méthode associée à un objet consiste en quelque sorte à faire "fonctionner" cet objet d'une manière particulière. Par exemple, on met en œuvre la méthode **methode4()** d'un objet **objet3**, à l'aide d'une instruction du type : **objet3.methode4()**
  (c.à.d. le nom de l'objet, puis le nom de la méthode, reliés l'un à l'autre par un point. Ce point joue un rôle essentiel : on peut le considérer comme un véritable ***opérateur***).

  Dans notre exemple, nous appliquons donc la méthode **append()** à l'objet **result**. Sous Python, les listes constituent un type particulier d'objets, auxquels on peut effectivement appliquer toute une série de méthodes. En l'occurrence, la méthode **append()** est donc une fonction spécifique des listes, qui sert à leur ajouter un élément par la fin. L'élément à ajouter est transmis entre les parenthèses, comme tout argument qui se respecte.

  Remarque :
  Nous aurions obtenu un résultat similaire si nous avions utilisé à la place de cette instruction une expression telle que "**result = result + [b]**". Cette façon de procéder est cependant moins rationnelle et moins efficace, car elle consiste à redéfinir à chaque itération de la boucle une ***nouvelle*** liste **result**, dans laquelle la totalité de la liste précédente est à chaque fois recopiée avec ajout d'un élément supplémentaire.
  Lorsque l'on utilise la méthode **append()**, par contre, l'ordinateur procède bel et bien à une modification de la liste existante (sans la recopier dans une nouvelle variable). Cette technique est préférable, car elle mobilise moins lourdement les ressources de l'ordinateur et elle est plus rapide (surtout lorsqu'il s'agit de traiter des listes volumineuses).

- Il n'est pas du tout indispensable que la valeur retournée par une fonction soit affectée à une variable (comme nous l'avons fait jusqu'ici dans nos exemples par souci de clarté).
  Ainsi, nous aurions pu tester les fonction cube() et table() en entrant les commandes :
  ```
  >>> print cube(9)
  >>> print table(9)
  >>> print table(9)[3]
  ```
  ou encore plus simplement encore :
  ```
  >>> cube(9)                                    ...    etc.
  ```

---

24 **return** peut également être utilisé sans aucun argument, à l'intérieur d'une fonction, pour provoquer sa fermeture immédiate. La valeur retournée dans ce cas est l'objet **None** (objet particulier, correspondant à "rien").

7.4 Utilisation des fonctions dans un script

Pour cette première approche des fonctions, nous n'avons utilisé jusqu'ici que le mode interactif de l'interpréteur Python.

Il est bien évident que les fonctions peuvent aussi s'utiliser dans des scripts. Veuillez donc essayer vous-même le petit programme ci-dessous, lequel calcule le volume d'une sphère à l'aide de la formule que vous connaissez certainement : $V = \frac{4}{3} \pi R^3$

```
def cube(n):
   return n**3

def volumeSphere(r):
   return 4 * 3.1416 * cube(r) / 3

r = input('Entrez la valeur du rayon : ')
print 'Le volume de cette sphère vaut', volumeSphere(r)
```

Notes :

A bien y regarder, ce programme comporte trois parties : les deux fonctions **cube()** et **volumeSphere()**, et ensuite le corps principal du programme.

Dans le corps principal du programme, il y a un appel de la fonction **volumeSphere()**.

A l'intérieur de la fonction **volumeSphere()**, il y a un appel de la fonction **cube()**.

Notez bien que les trois parties du programme ont été disposées dans un certain ordre : d'abord la définition des fonctions, et ensuite le corps principal du programme. Cette disposition est nécessaire, parce que l'interpréteur exécute les lignes d'instructions du programme l'une après l'autre, dans l'ordre où elles apparaissent dans le code source. Dans le script, la définition des fonctions doit donc précéder leur utilisation.

Pour vous en convaincre, intervertissez cet ordre (en plaçant par exemple le corps principal du programme au début), et prenez note du type de message d'erreur qui est affiché lorsque vous essayez d'exécuter le script ainsi modifié.

En fait, le corps principal d'un programme Python constitue lui-même une entité un peu particulière, qui est toujours reconnue dans le fonctionnement interne de l'interpréteur sous le nom réservé **__main__** (le mot "*main*" signifie "principal", en anglais. Il est encadré par des caractères "souligné" en double, pour éviter toute confusion avec d'autres symboles). L'exécution d'un script commence toujours avec la première instruction de cette entité **__main__**, où qu'elle puisse se trouver dans le listing. Les instructions qui suivent sont alors exécutées l'une après l'autre, dans l'ordre, jusqu'au premier appel de fonction. Un appel de fonction est comme un détour dans le flux de l'exécution : au lieu de passer à l'instruction suivante, l'interpréteur exécute la fonction appelée, puis revient au programme appelant pour continuer le travail interrompu. Pour que ce mécanisme puisse fonctionner, il faut que l'interpréteur ait pu lire la définition de la fonction ***avant*** l'entité **__main__**, et celle-ci sera donc placée en général à la fin du script.

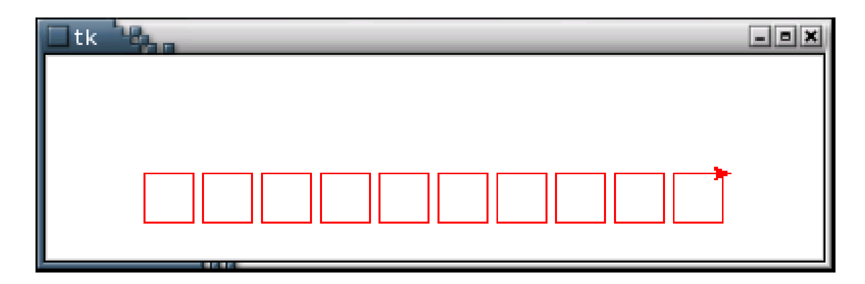
Dans notre exemple, l'entité **__main__** appelle une première fonction qui elle-même en appelle une deuxième. Cette situation est très fréquente en programmation. Si vous voulez comprendre correctement ce qui se passe dans un programme, vous devez donc apprendre à lire un script, non pas de la première à la dernière ligne, mais plutôt en suivant un cheminement analogue à ce qui se passe lors de l'exécution de ce script. Cela signifie concrètement que vous devrez souvent analyser un script en commençant par ses dernières lignes !

7.5 Modules de fonctions

Afin que vous puissiez mieux comprendre encore la distinction entre la définition d'une fonction et son utilisation au sein d'un programme, nous vous suggérons de placer fréquemment vos définitions de fonctions dans un module Python, et le programme qui les utilise dans un autre.

Exemple :

On souhaite réaliser la série de dessins ci-dessous, à l'aide du module *turtle :*



Écrivez les lignes de code suivantes, et sauvegardez-les dans un fichier auquel vous donnerez le nom ***dessins_tortue.py*** :

```
from turtle import *

def carre(taille, couleur):
    "fonction qui dessine un carré de taille et de couleur déterminées"
    color(couleur)
    c =0
    while c <4:
        forward(taille)
        right(90)
        c = c +1
```

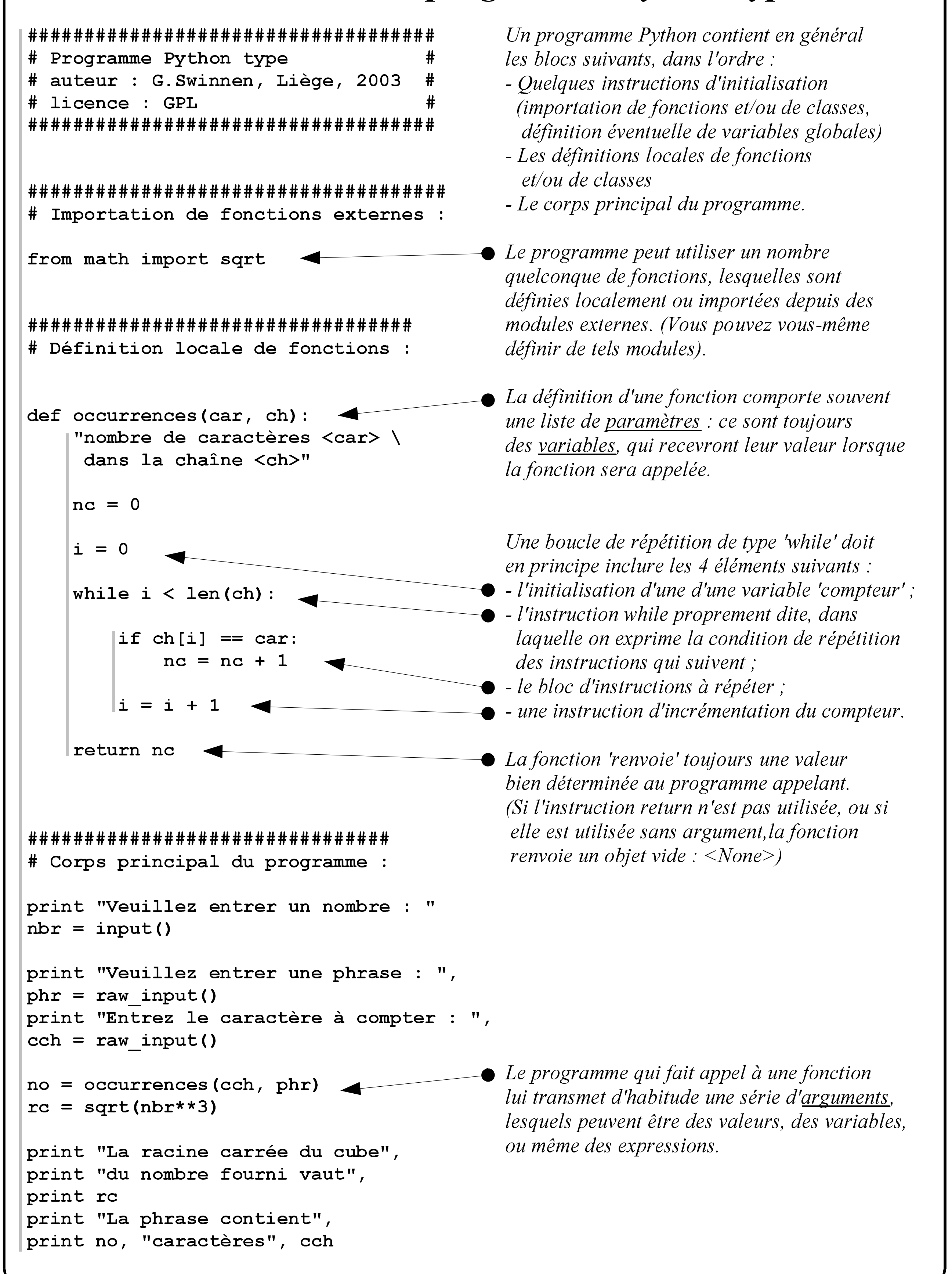
Ce fichier est dorénavant un véritable **module** de fonctions Python, au même titre que les modules *turtle* ou *math* que vous connaissez déjà. Vous pouvez donc l'utiliser dans n'importe quel autre script, comme celui-ci, par exemple, qui effectuera le travail demandé :

```
from dessins_tortue import *

up()                    # relever le crayon
goto(-150, 50)          # reculer en haut à gauche

# dessiner dix carrés rouges, alignés :
i = 0
while i < 10:
    down()              # abaisser le crayon
    carre(25, 'red')    # tracer un carré
    up()                # relever le crayon
    forward(30)         # avancer + loin
    i = i +1

a = input()             # attendre
```



Résumé : Structure d'un programme Python type

```
####################################
# Programme Python type            #
# auteur : G.Swinnen, Liège, 2003  #
# licence : GPL                    #
####################################


#####################################
# Importation de fonctions externes :

from math import sqrt

###################################
# Définition locale de fonctions :


def occurrences(car, ch):
    "nombre de caractères <car> \
     dans la chaîne <ch>"

    nc = 0

    i = 0

    while i < len(ch):

        if ch[i] == car:
            nc = nc + 1

        i = i + 1

    return nc


################################
# Corps principal du programme :

print "Veuillez entrer un nombre : "
nbr = input()

print "Veuillez entrer une phrase : ",
phr = raw_input()
print "Entrez le caractère à compter : ",
cch = raw_input()

no = occurrences(cch, phr)
rc = sqrt(nbr**3)

print "La racine carrée du cube",
print "du nombre fourni vaut",
print rc
print "La phrase contient",
print no, "caractères", cch
```

Un programme Python contient en général les blocs suivants, dans l'ordre :
- Quelques instructions d'initialisation (importation de fonctions et/ou de classes, définition éventuelle de variables globales)
- Les définitions locales de fonctions et/ou de classes
- Le corps principal du programme.

Le programme peut utiliser un nombre quelconque de fonctions, lesquelles sont définies localement ou importées depuis des modules externes. (Vous pouvez vous-même définir de tels modules).

La définition d'une fonction comporte souvent une liste de paramètres : ce sont toujours des variables, qui recevront leur valeur lorsque la fonction sera appelée.

Une boucle de répétition de type 'while' doit en principe inclure les 4 éléments suivants :
- l'initialisation d'une d'une variable 'compteur' ;
- l'instruction while proprement dite, dans laquelle on exprime la condition de répétition des instructions qui suivent ;
- le bloc d'instructions à répéter ;
- une instruction d'incrémentation du compteur.

La fonction 'renvoie' toujours une valeur bien déterminée au programme appelant. (Si l'instruction return n'est pas utilisée, ou si elle est utilisée sans argument,la fonction renvoie un objet vide : <None>)

Le programme qui fait appel à une fonction lui transmet d'habitude une série d'arguments, lesquels peuvent être des valeurs, des variables, ou même des expressions.

Exercices :

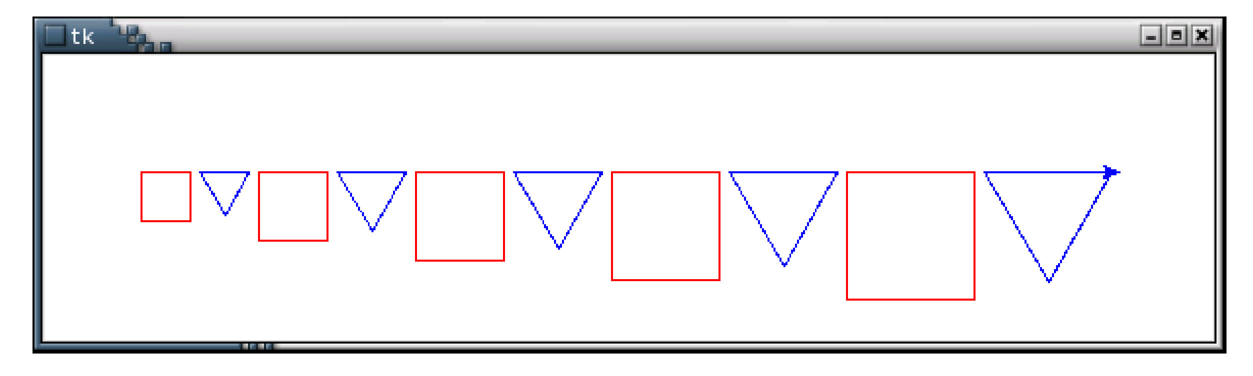
e 44. Définissez une fonction **ligneCar(n, ca)** qui renvoie une chaîne de **n** caractères **ca**.

e 45. Définissez une fonction **surfCercle(R)**. Cette fonction doit renvoyer la surface (l'aire) d'un cercle dont on lui a fourni le rayon R en argument. Par exemple, l'exécution de l'instruction :
`print surfCercle(2.5)` doit donner le résultat **19.635**

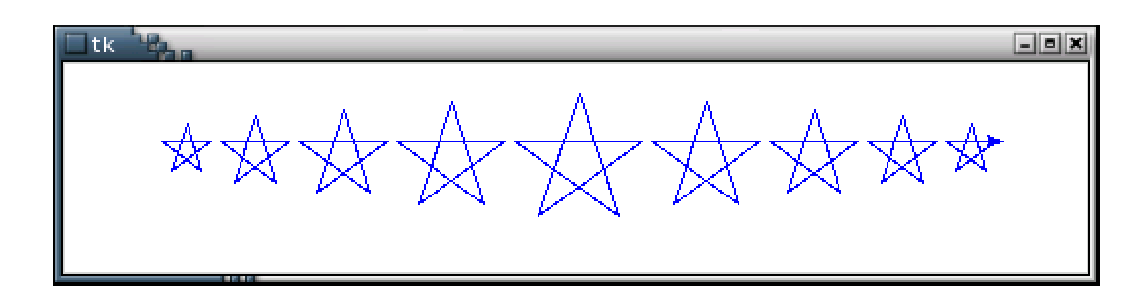
e 46. Définissez une fonction **volBoite(x1,x2,x3)** qui renvoie le volume d'une boîte parallélipipédique dont on fournit les trois dimensions **x1**, **x2**, **x3** en arguments.
Par exemple, l'exécution de l'instruction :
`print volBoite(5.2, 7.7, 3.3)` doit donner le résultat : **132.13**

e 47. Définissez une fonction **maximum(n1,n2,n3)** qui renvoie le plus grand de 3 nombres **n1**, **n2**, **n3** fournis en arguments. Par exemple, l'exécution de l'instruction :
`print maximum(2,5,4)` doit donner le résultat : **5**

e 48. Complétez le module de fonctions graphiques ***dessins_tortue.py*** décrit à la page 72.
Commencez par ajouter un paramètre **angle** à la fonction **carre()**, de manière à ce que les carrés puissent être tracés dans différentes orientations.
Définissez ensuite une fonction **triangle(taille, couleur, angle)** capable de dessiner un triangle équilatéral d'une taille, d'une couleur et d'une orientation bien déterminées.
Testez votre module à l'aide d'un programme qui fera appel à ces fonctions à plusieurs reprises, avec des arguments variés pour dessiner une série de carrés et de triangles :

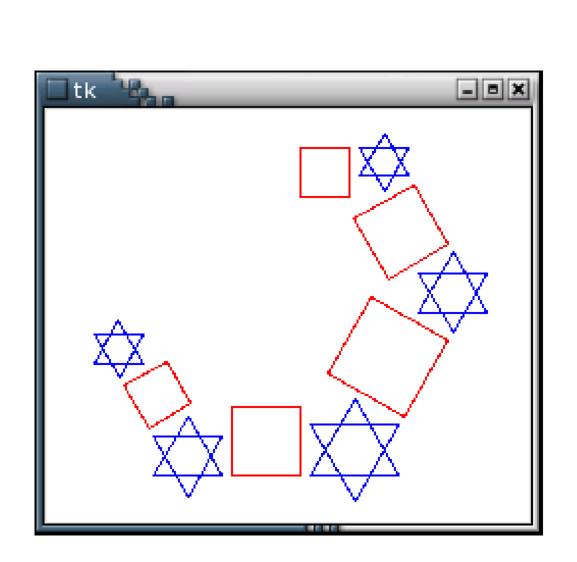


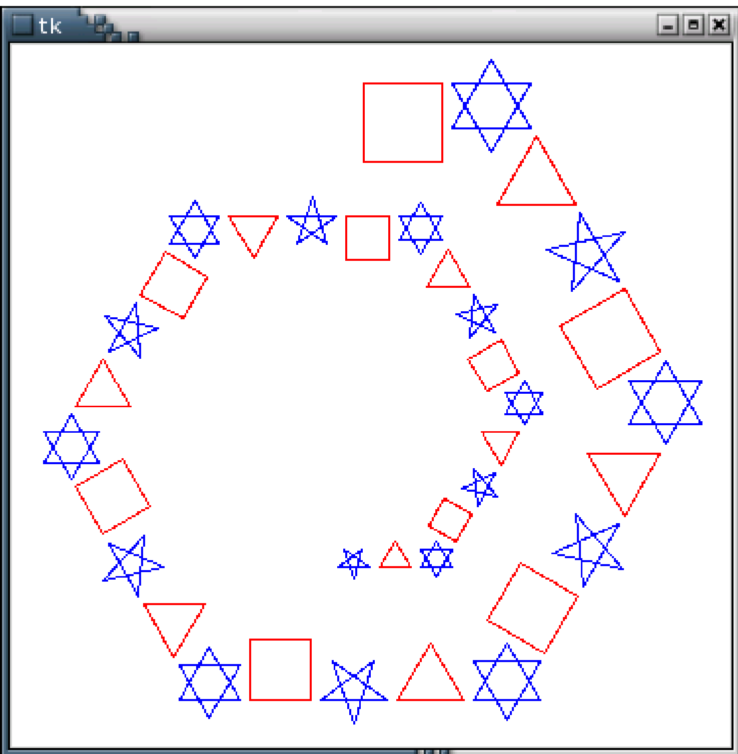
e 49. Ajoutez au module de l'exercice précédent une fonction **etoile5()** spécialisée dans le dessin d'étoiles à 5 branches. Dans votre programme principal, insérez une boucle qui dessine une



rangée horizontale de de 9 petites étoiles de tailles variées :

e 50. Ajoutez au module de l'exercice précédent une fonction **etoile6()** capable de dessiner une étoile à 6 branches, elle-même constituée de deux triangles équilatéraux imbriqués. Cette nouvelle fonction devra faire appel à la fonction **triangle()** définie précédemment.
Votre programme principal dessinera également une série de ces étoiles :





e 51. Définissez une fonction **compteCar(ca,ch)** qui renvoie le nombre de fois que l'on rencontre le caractère **ca** dans la chaîne de caractères **ch**. Par exemple, l'exécution de l'instruction :
`print compteCar('e','Cette phrase est un exemple')` doit donner le résultat : `7`

e 52. Définissez une fonction **indexMax(liste)** qui renvoie l'index de l'élément ayant la valeur la plus élevée dans la liste transmise en argument. Exemple d'utilisation :

```
serie = [5, 8, 2, 1, 9, 3, 6, 7]
print indexMax(serie)
4
```

e 53. Définissez une fonction **nomMois(n)** qui renvoie le nom du n^e mois de l'année.
Par exemple, l'exécution de l'instruction :
`print nomMois(4)` doit donner le résultat : `Avril`

e 54. Définissez une fonction **inverse(ch)** qui permette d'inverser les l'ordre des caractères d'une chaîne quelconque. (La chaîne inversée sera renvoyée au programme appelant).

e 55. Définissez une fonction **compteMots(ph)** qui renvoie le nombre de mots contenus dans la phrase **ph** (On considère comme mots les ensembles de caractères inclus entre des espaces).

7.6 Typage des paramètres

Vous avez appris que le typage des variables sous Python est un typage dynamique, ce qui signifie que le type d'une variable est défini au moment où on lui affecte une valeur. Ce mécanisme fonctionne aussi pour les paramètres d'une fonction. Le type d'un paramètre sera le même que celui de l'argument qui aura été transmis à la fonction. Exemple :

```
>>> def afficher3fois(arg):
...     print arg, arg, arg
...

>>> afficher3fois(5)
5 5 5

>>> afficher3fois('zut')
zut zut zut

>>> afficher3fois([5, 7])
[5, 7] [5, 7] [5, 7]

>>> afficher3fois(6**2)
36 36 36
```

Dans cet exemple, vous pouvez constater que la même fonction **afficher3fois()** accepte dans tous les cas l'argument qu'on lui transmet, que cet argument soit un nombre, une chaîne de caractères, une liste, ou même une expression. Dans ce dernier cas, Python commence par évaluer l'expression, et c'est le résultat de cette évaluation qui est transmis comme argument à la fonction.

7.7 Valeurs par défaut pour les paramètres

Dans la définition d'une fonction, il est possible (et souvent souhaitable) de définir un argument par défaut pour chacun des paramètres. ***On obtient ainsi une fonction qui peut être appelée avec une partie seulement des arguments attendus***. Exemples :

```
>>> def politesse(nom, vedette ='Monsieur'):
...    print "Veuillez agréer ,", vedette, nom, ", mes salutations distinguées."
...

>>> politesse('Dupont')
Veuillez agréer , Monsieur Dupont , mes salutations distinguées.

>>> politesse('Durand', 'Mademoiselle')
Veuillez agréer , Mademoiselle Durand , mes salutations distinguées.
```

Lorsque l'on appelle cette fonction en ne lui fournissant que le premier argument, le second reçoit tout de même une valeur par défaut. Si l'on fournit les deux arguments, la valeur par défaut pour le deuxième est tout simplement ignorée.

Vous pouvez définir une valeur par défaut pour tous les paramètres, ou une partie d'entre eux seulement. Dans ce cas, cependant, les paramètres sans valeur par défaut doivent précéder les autres dans la liste. Par exemple, la définition ci-dessous est incorrecte :

```
>>> def politesse(vedette ='Monsieur', nom):
```

Autre exemple :

```
>>> def question(annonce, essais =4, please ='Oui ou non, s.v.p.!'):
...     while essais >0:
...         reponse = raw_input(annonce)
...         if reponse in ('o', 'oui','O','Oui','OUI'):
...             return 1
...         if reponse in ('n','non','N','Non','NON'):
...             return 0
...         print please
...         essais = essais-1
...
>>>
```

Cette fonction peut être appelée de différentes façons, telles par exemple :

`rep = question('Voulez-vous vraiment terminer ? ')` ou bien :

`rep = question('Faut-il effacer ce fichier ? ', 3)` ou même encore :

`rep = question('Avez-vous compris ? ', 2, 'Répondez par oui ou par non !')`

(Prenez la peine d'essayer et de décortiquer cet exemple)

7.8 Arguments avec étiquettes

Dans la plupart des langages de programmation, les arguments que l'on fournit lors de l'appel d'une fonction doivent être fournis ***exactement dans le même ordre*** que celui des paramètres qui leur correspondent dans la définition de la fonction.

Python autorise cependant une souplesse beaucoup plus grande. Si les paramètres annoncés dans la définition de la fonction ont reçu chacun une valeur par défaut, sous la forme déjà décrite ci-dessus, on peut faire appel à la fonction en fournissant les arguments correspondants ***dans n'importe quel ordre, à la condition de désigner nommément les paramètres*** correspondants. Exemple :

```
>>> def oiseau(voltage=100, etat='allumé', action='danser la java'):
...     print 'Ce perroquet ne pourra pas', action
...     print 'si vous le branchez sur', voltage, 'volts !'
...     print "L'auteur de ceci est complètement", etat
...

>>> oiseau(etat='givré', voltage=250, action='vous approuver')
Ce perroquet ne pourra pas vous approuver
si vous le branchez sur 250 volts !
L'auteur de ceci est complètement givré

>>> oiseau()
Ce perroquet ne pourra pas danser la java
si vous le branchez sur 100 volts !
L'auteur de ceci est complètement allumé
```

Exercices :

e 56. Modifiez la fonction **volBoite(x1,x2,x3)** que vous avez définie dans un exercice précédent, de manière à ce qu'elle puisse être appelée avec trois, deux, un seul, ou même aucun argument. Utilisez pour ceux ci des valeurs par défaut égales à) 10.
Par exemple :

`print volBoite()` doit donner le résultat : **1000**
`print volBoite(5.2)` doit donner le résultat : **520.0**
`print volBoite(5.2, 3)` doit donner le résultat : **156.0**

e 57. Modifiez la fonction **volBoite(x1,x2,x3)** ci-dessus de manière à ce qu'elle puisse être appelée avec un, deux, ou trois arguments. Si un seul est utilisé, la boîte est considérée comme cubique (l'argument étant l'arête de ce cube). Si deux sont utilisés, la boîte est considérée comme un prisme à base carrée. (Dans ce cas le premier argument est le côté du carré, et le second la hauteur du prisme). Si trois arguments sont utilisés, la boîte est considérée comme un parallélépipède. Par exemple :

`print volBoite()` doit donner le résultat : **-1** (→ indication d'une erreur).
`print volBoite(5.2)` doit donner le résultat : **140.608**
`print volBoite(5.2, 3)` doit donner le résultat : **81.12**
`print volBoite(5.2, 3, 7.4)` doit donner le résultat : **115.44**

e 58. Définissez une fonction **changeCar(ch,ca1,ca2,debut,fin)** qui remplace tous les caractères **ca1** par des caractères **ca2** dans la chaîne de caractères **ch**, à partir de l'indice **debut** et jusqu'à l'indice **fin**, ces deux derniers arguments pouvant être omis (et dans ce cas la chaîne est traitée d'une extrémité à l'autre). Exemples de la fonctionnalité attendue :

```
>>> phrase = 'Ceci est une toute petite phrase.'
>>> print changeCar(phrase, ' ', '*')
Ceci*est*une*toute*petite*phrase.
>>> print changeCar(phrase, ' ', '*', 8, 12)
Ceci est*une*toute petite phrase.
>>> print changeCar(phrase, ' ', '*', 12)
Ceci est une*toute*petite*phrase.
>>> print changeCar(phrase, ' ', '*', fin = 12)
Ceci*est*une*toute petite phrase.
```

e 59. Définissez une fonction **eleMax(liste,debut,fin)** qui retourne l'élément ayant la plus grande valeur dans la **liste** transmise. Les deux arguments **debut** et **fin** indiqueront les indices entre lesquels doit s'exercer la recherche, et chacun d'eux pourra être omis (comme dans l'exercice précédent). Exemples de la fonctionnalité attendue :

```
>>> serie = [9, 3, 6, 1, 7, 5, 4, 8, 2]
>>> print eleMax(serie)
9
>>> print eleMax(serie, 2, 5)
7
>>> print eleMax(serie, 2)
8
>>> print eleMax(serie, fin =3, debut =1)
6
```

Chapitre 8 : Utilisation de fenêtres et de graphismes

8.1 Interfaces graphiques (GUI)

Jusqu'à présent, nous avons utilisé Python exclusivement "en mode texte". Nous avons procédé ainsi parce qu'il nous fallait absolument d'abord dégager un certain nombre de concepts élémentaires ainsi que la structure de base du langage, avant d'envisager des expériences impliquant des objets informatiques plus élaborés (fenêtres, images, sons, etc.). Nous pouvons à présent nous permettre une petite incursion dans le vaste domaine des interfaces graphiques, mais ce ne sera qu'un premier amuse-gueule : il nous reste en effet encore bien des choses fondamentales à apprendre, et pour nombre d'entre elles l'approche textuelle reste la plus abordable.

Si vous ne le saviez pas encore, apprenez dès à présent que le domaine des interfaces graphiques (ou *GUI : Graphical User Interface*) est extrêmement complexe. Chaque système d'exploitation peut en effet proposer plusieurs "bibliothèques" de fonctions graphiques de base, auxquelles viennent fréquemment s'ajouter de nombreux compléments, plus ou moins spécifiques de langages de programmation particuliers. Tous ces composants sont généralement présentés comme des ***classes d'objets***, dont il vous faudra étudier les ***attributs*** et les ***méthodes***.

Avec Python, la bibliothèque graphique la plus utilisée jusqu'à présent est la bibliothèque ***Tkinter***, qui est une adaptation de la bibliothèque ***Tk*** développée à l'origine pour le langage ***Tcl***. Plusieurs autres bibliothèques graphiques fort intéressantes ont été proposées pour Python : ***wxPython***, **pyQT**, **pyGTK**, etc. Il existe également des possibilités d'utiliser les bibliothèques de *widgets* ***Java*** et les MFC de *Window$*.
Dans le cadre de ces notes, nous nous limiterons cependant à *Tkinter*, dont il existe fort heureusement des versions similaires (et gratuites) pour les plates-formes *Linux*, *Window$* et *M@c*.

8.2 Premiers pas avec Tkinter

Pour la suite des explications, nous supposerons bien évidemment que le module *Tkinter* a déjà été installé sur votre système. Pour pouvoir en utiliser les fonctionnalités dans un script Python, il faut que l'une des premières lignes de ce script contienne l'instruction d'importation :

```
from Tkinter import *
```



Comme toujours sous Python, il n'est même pas nécessaire d'écrire un script. Vous pouvez faire un grand nombre d'expériences directement à la ligne de commande, en ayant simplement lancé Python en mode interactif. Dans l'exemple qui suit, nous allons créer une fenêtre très simple, et y ajouter deux *widgets*[25] typiques : un bout de texte (ou *label*) et un bouton (ou *button*).

```
>>> from Tkinter import *
>>> fen1 = Tk()
>>> tex1 = Label(fen1, text='Bonjour tout le monde !', fg='red')
>>> tex1.pack()
>>> bou1 = Button(fen1, text='Quitter', command = fen1.destroy)
>>> bou1.pack()
>>> fen1.mainloop()
```

25 "*widget*" est le résultat de la contraction de l'expression "*window gadget*". Dans certains environnements de programmation, on appellera cela plutôt un "contrôle" ou un "composant graphique". Ce terme désigne en fait toute entité susceptible d'être placée dans une fenêtre d'application, comme par exemple un bouton, une case à cocher, une image, etc., et parfois aussi la fenêtre elle-même.

Note : Suivant la version de Python utilisée, vous verrez déjà apparaître la fenêtre d'application immédiatement après avoir entré la deuxième commande de cet exemple, ou bien seulement après la septième[26].

Examinons à présent plus en détail chacune des lignes de commandes exécutées :

1. Comme cela a déjà été expliqué précédemment, il est aisé de construire différents *modules* Python, qui contiendront des scripts, des définitions de fonctions, des classes d'objets, etc. On peut alors importer tout ou partie de ces modules dans n'importe quel programme, ou même dans l'interpréteur fonctionnant en mode interactif (c.à.d. directement à la ligne de commande). C'est ce que nous faisons à la première ligne de notre exemple : "`from Tkinter import *`" consiste à importer toutes les ***classes*** contenues dans le module *Tkinter*.
   Nous devrons de plus en plus souvent parler de ces ***classes***. En programmation, on appelle ainsi des ***générateurs d'objets***, lesquels sont eux-mêmes des morceaux de programmes réutilisables.
   Nous n'allons pas essayer de vous fournir dès à présent une définition définitive et précise de ce que sont les objets et les classes, mais plutôt vous proposer d'en utiliser directement quelques-un(e)s. Nous affinerons notre compréhension petit à petit par la suite.

2. A la deuxième ligne de notre exemple : "`fen1 = Tk()`", nous utilisons l'une des classes du module *Tkinter*, la classe **Tk()**, et nous en créons une ***instance*** (autre terme désignant un ***objet*** spécifique), à savoir la fenêtre **fen1**.

   Ce processus ***d'instanciation d'un objet à partir d'une classe*** est une opération fondamentale dans les techniques actuelles de programmation. Celles-ci font en effet de plus en plus souvent appel à une méthodologie que l'on appelle *"programmation orientée objet" (ou OOP : Object Oriented Programming)*.

   La ***classe*** est en quelque sorte un modèle général (ou un moule) à partir duquel on demande à la machine de construire un ***objet*** informatique particulier. La classe contient toute une série de définitions et d'options diverses, dont nous n'utilisons qu'une partie dans l'objet que nous créons à partir d'elle. Ainsi la classe **Tk()** , qui est l'une des classes les plus fondamentales de la bibliothèque *Tkinter*, contient tout ce qu'il faut pour engendrer différents types de fenêtres d'application, de tailles ou de couleurs diverses, avec ou sans barre de menus, etc.

   Nous nous en servons ici pour créer notre objet graphique de base, à savoir la fenêtre qui contiendra tout le reste. Dans les parenthèses de **Tk()**, nous pourrions préciser différentes options, mais nous laisserons cela pour un peu plus tard.

   L'instruction d'instanciation ressemble à une simple affectation de variable. Comprenons bien cependant qu'il se passe ici deux choses à la fois :

   - ***la création d'un nouvel objet***, (lequel peut être complexe et donc occuper un espace mémoire considérable)
   - ***l'affectation d'une variable*** qui va désormais servir de référence pour manipuler l'objet[27].

---

26 Si vous effectuez cet exercice sous *Window$*, nous vous conseillons d'utiliser de préférence une version standard de Python dans une fenêtre DOS ou dans IDLE plutôt que *PythonWin.* Vous pourrez mieux observer ce qui se passe après l'entrée de chaque commande.

27 Cette concision du langage est une conséquence du typage dynamique des variables en vigueur sous Python. D'autres langages utilisent une instruction particulière (telle que **new**) pour instancier un nouvel objet. Exemple :
maVoiture = new Cadillac (instanciation d'un objet de classe **Cadillac**, référencé dans la variable **maVoiture**)

3. A la troisième ligne : "**tex1 = Label(fen1, text='Bonjour tout le monde !', fg='red')**", nous créons un autre objet (un *widget*), cette fois à partir de la classe **Label()**.
   Comme son nom l'indique, cette classe définit toutes sortes d'*étiquettes* (ou de "*libellés*"). En fait, il s'agit tout simplement de fragments de texte quelconques, utilisables pour afficher des informations et des messages divers à l'intérieur d'une fenêtre.
   Nous efforçant d'apprendre au passage la manière correcte d'exprimer les choses, nous dirons donc que nous créons ici l'objet **tex1** par ***instanciation*** de la classe **Label()**.

   Remarquons ici que nous faisons appel à une classe, de la même manière que nous faisons appel à une fonction : c.à.d. en fournissant un certain nombre d'arguments dans des parenthèses. Nous verrons plus loin qu'une classe est en fait une sorte de 'conteneur' dans lequel sont regroupées des fonctions et des données.

   Quels arguments avons-nous donc fournis pour cette instanciation ?

   - Le premier argument transmis (**fen1**), indique que le nouveau *widget* que nous sommes en train de créer ***sera contenu dans un autre widget préexistant***, que nous désignons donc ici comme son "maître" : l'objet **fen1** est le *widget* ***maître*** de l'objet **tex1**. (On pourra dire aussi que l'objet **tex1** est un *widget* ***esclave*** de l'objet **fen1**).

   - Les deux arguments suivants servent à préciser la forme exacte que doit prendre notre *widget*. Ce sont en effet deux ***options*** de création, chacune fournie sous la forme d'une chaîne de caractères : d'abord le texte de l'étiquette, ensuite sa couleur d'avant-plan (ou *foreground*, en abrégé **fg**). Ainsi le texte que nous voulons afficher est bien défini, et il doit apparaître coloré en rouge.
     Nous pourrions encore préciser bien d'autres caractéristiques : la police à utiliser, ou la couleur d'arrière-plan, par exemple. Toutes ces caractéristiques ont cependant une valeur par défaut dans les définitions internes de la classe **Label().** Nous ne devons indiquer des options que pour les caractéristiques que nous souhaitons différentes du modèle standard.

4. A la quatrième ligne de notre exemple : "**tex1.pack()**" , nous activons une ***méthode*** associée à l'objet **tex1** : la méthode **pack()**. Nous avons déjà rencontré ce terme de *méthode* (à propos des listes, notamment). Une méthode est une fonction intégrée à un objet (on dira aussi qu'elle est ***encapsulée*** dans l'objet). Nous apprendrons bientôt qu'un objet informatique est en fait un morceau de programme contenant toujours :

   - un certain nombre de données (numériques ou autres), contenues dans des variables de types divers : on les appelle les ***attributs*** (ou les propriétés) de l'objet.
   - un certain nombre de procédures ou de fonctions (qui sont donc des algorithmes) : on les appelle les ***méthodes*** de l'objet.

   La méthode **pack()** fait partie d'un ensemble de méthodes qui sont applicables non seulement aux *widgets* de la classe **Label()**, mais aussi à la plupart des autres *widgets Tkinter*, et qui agissent sur leur disposition géométrique dans la fenêtre. Comme vous pouvez le constater par vous-même si vous entrez les commandes de notre exemple une par une, la méthode **pack()** réduit automatiquement la taille de la fenêtre "maître" afin qu'elle soit juste assez grande pour contenir les *widgets* "esclaves" définis au préalable.

5. A la cinquième ligne : "**bou1 = Button(fen1, text='Quitter', command = fen1.destroy)**", nous créons notre second *widget* "esclave" : un bouton.
   Comme nous l'avons fait pour le widget précédent, nous appelons la classe **Button()** en fournissant entre parenthèses un certain nombre d'arguments. Étant donné qu'il s'agit cette fois d'un objet interactif, nous devons préciser avec l'option **command** ce qui devra se passer lorsque l'utilisateur effectuera un clic sur le bouton. Dans ce cas précis, nous actionnerons la méthode **destroy** associée à l'objet **fen1**, ce qui devrait provoquer l'effacement de la fenêtre.

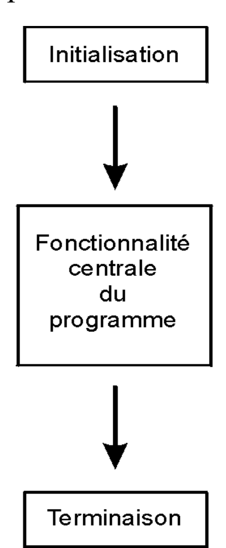
6. La sixième ligne utilise la méthode **pack()** pour adapter la géométrie de la fenêtre au nouvel objet que nous venons d'y intégrer.

7. La septième ligne : "**fen1.mainloop()**" est très importante, parce que c'est elle qui provoque le démarrage du ***réceptionnaire d'événements*** associé à la fenêtre. Cette instruction est nécessaire pour que votre application soit "à l'affût" des clics de souris, des pressions exercées sur les touches du clavier, etc. C'est donc cette instruction qui "la met en marche", en quelque sorte.

   Comme son nom l'indique (*mainloop*), il s'agit d'une méthode de l'objet **fen1**, qui active une ***boucle*** de programme, laquelle "tournera" en permanence en tâche de fond, dans l'attente de messages émis par le système d'exploitation de l'ordinateur. Celui-ci interroge en effet sans cesse son environnement, notamment au niveau des périphériques d'entrée (souris, clavier, etc.). Lorsqu'un événement quelconque est détecté, divers messages décrivant cet événement sont expédiés aux programmes qui souhaitent en être avertis. Voyons cela un peu plus en détail.

8.3 Programmes pilotés par des événements

Vous venez d'expérimenter votre premier programme utilisant une interface graphique. Ce type de programme est structuré d'une manière différente des scripts "textuels" étudiés auparavant.

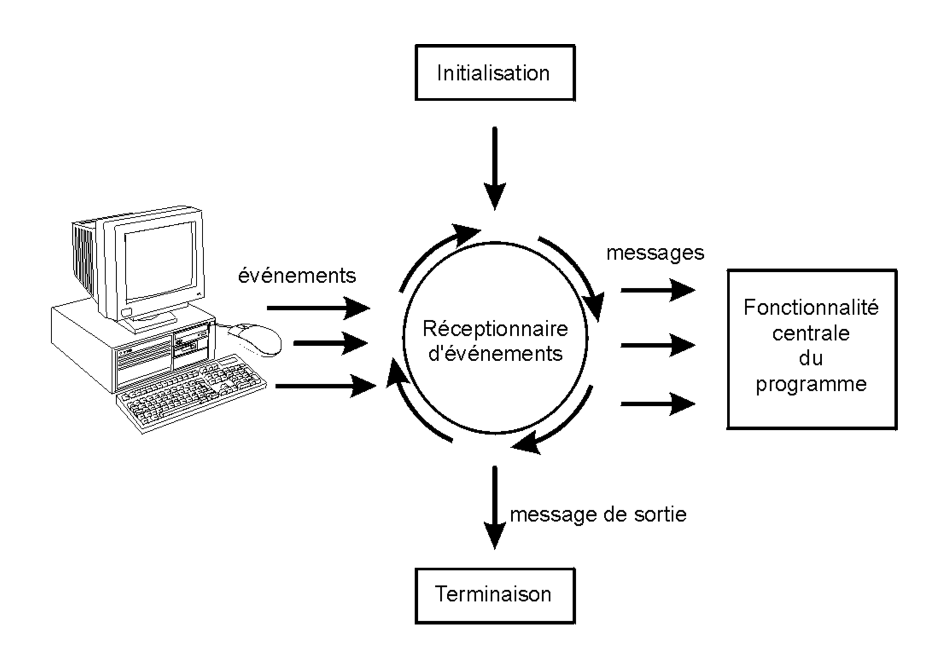


Tous les programmes d'ordinateur comportent grosso-modo trois phases principales : ***une phase d'initialisation***, laquelle contient les instructions qui préparent le travail à effectuer (appel des modules externes nécessaires, ouverture de fichiers, connexion à un serveur de bases de données ou à l'internet, etc.), ***une phase centrale*** où l'on trouve la véritable fonctionnalité du programme (c.à.d. tout ce qu'il est censé faire : afficher des données à l'écran, effectuer des calculs, modifier le contenu d'un fichier, imprimer, etc.), et enfin ***une phase de terminaison*** qui sert à clôturer "proprement" les opérations (c.à.d fermer les fichiers restés ouverts, couper les connexions externes, etc.)

Dans un programme "en mode texte", ces trois phases sont simplement organisées suivant un schéma linéaire comme dans l'illustration ci-contre. En conséquence, ces programmes se caractérisent par une interactivité très limitée avec l'utilisateur. Celui-ci ne dispose pratiquement d'aucune liberté : il lui est demandé de temps à autre d'entrer des données au clavier, mais toujours dans un ordre prédéterminé correspondant à la séquence d'instructions du programme.

Dans le cas d'un programme qui utilise une interface graphique, par contre, l'organisation interne est différente. On dit d'un tel programme qu'il est ***piloté par les événements***. Après sa phase d'initialisation, un programme de ce type se met en quelque sorte "en attente", et passe la main à un autre logiciel, lequel est plus ou moins intimement intégré au système d'exploitation de l'ordinateur et "tourne" en permanence.

Ce ***réceptionnaire d'événements*** scrute sans cesse tous les périphériques (clavier, souris, horloge, modem, etc.) et réagit immédiatement lorsqu'un événement y est détecté.
Un tel événement peut être une action quelconque de l'utilisateur : déplacement de la souris, appui sur une touche, etc., mais aussi un événement externe ou un automatisme (top d'horloge, par ex.)



Lorsqu'il détecte un événement, le réceptionnaire envoie un message spécifique au programme[28], lequel doit être conçu pour réagir en conséquence.

La phase d'initialisation d'un programme utilisant une interface graphique comporte un ensemble d'instructions qui mettent en place les divers composants interactifs de cette interface (fenêtres, boutons, cases à cocher, etc.). D'autres instructions définissent les messages d'événements qui devront être pris en charge : on peut en effet décider que le programme ne réagira qu'à certains événements en ignorant tous les autres.

Alors que dans un programme "textuel", la phase centrale est constituée d'une suite d'instructions qui décrivent à l'avance l'ordre dans lequel la machine devra exécuter ses différentes tâches (même s'il est prévu des cheminements différents en réponse à certaines conditions rencontrées en cours de route), on ne trouve dans la phase centrale d'un programme avec interface graphique qu'un ensemble de fonctions indépendantes. Chacune de ces fonctions est appelée spécifiquement lorsqu'un événement particulier est détecté par le système d'exploitation : elle effectue alors le travail que l'on attend du programme en réponse à cet événement, et rien d'autre[29].

Il est important de bien comprendre ici que pendant tout ce temps, le réceptionnaire continue à "tourner" et à guetter l'apparition d'autres événements éventuels.

S'il arrive d'autres événements, il peut donc se faire qu'une seconde fonction (ou une 3e, une 4e, ...) soit activée et commence à effectuer son travail "en parallèle" avec la première qui n'a pas encore terminé le sien[30]. Les systèmes d'exploitation et les langages modernes permettent en effet ce parallélisme que l'on appelle aussi ***multitâche***.

Au chapitre précédent de ces notes, nous vous avons déjà fait remarquer que la structure du texte d'un programme n'indique pas directement l'ordre dans lequel les instructions seront finalement exécutées. Cette remarque s'applique encore bien davantage dans le cas d'un programme avec interface graphique, puisque l'ordre dans lequel les fonctions sont appelées n'est plus inscrit nulle part dans le programme. Ce sont les événements qui pilotent !

Tout ceci doit vous paraître un peu compliqué. Nous allons l'illustrer dans quelques exemples.

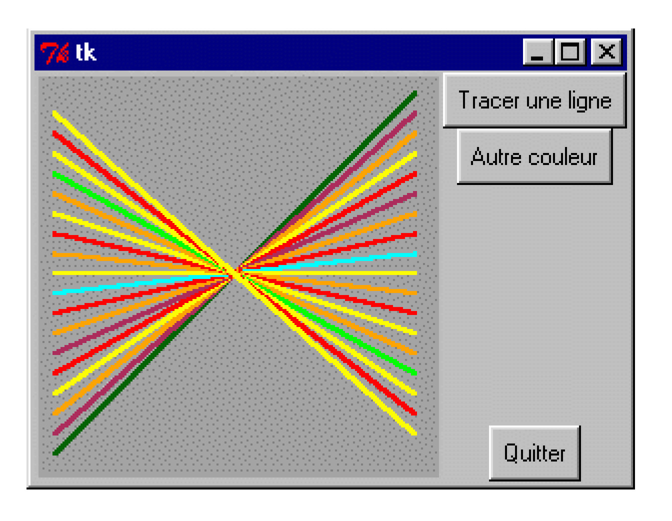
28 Ces messages sont souvent notés WM (Window messages) dans un environnement graphique constitué de fenêtres (avec de nombreuses zones réactives : boutons, cases à cocher, menus déroulants, etc.). Dans la description des algorithmes, il arrive fréquemment aussi qu'on confonde ces messages avec les événements eux-mêmes.

29 Au sens strict, une telle fonction qui ne devra retourner aucune valeur est donc plutôt une ***procédure*** (cfr. page 69).

30 En particulier, la même fonction peut être appelée plusieurs fois en réponse à l'occurrence de quelques événements identiques, la même tâche étant alors effectuée en plusieurs exemplaires concurrents. Nous verrons plus loin qu'il peut en résulter des "effets de bords" gênants.

8.3.1 Exemple graphique : tracé de lignes dans un canevas

Le script décrit ci-dessous crée une fenêtre comportant trois boutons et un ***canevas***. Suivant la terminologie de *Tkinter*, un canevas est une surface rectangulaire délimitée, dans laquelle on peut installer ensuite divers dessins et images à l'aide de méthodes spécifiques[31].



Lorsque l'on actionne le bouton "Tracer une ligne", une nouvelle ligne colorée apparaît sur le canevas, avec à chaque fois une inclinaison différente de la précédente.

Si l'on actionne le bouton "Autre couleur", une nouvelle couleur est tirée au hasard dans une série limitée. Cette couleur est celle qui s'appliquera aux tracés suivants.

Le bouton "Quitter" sert bien évidemment à terminer l'application en refermant la fenêtre.

```
# Petit exercice utilisant la bibliothèque graphique Tkinter

# --- définition des fonctions gestionnaires d'événements : ---
def drawline():
    "Tracé d'une ligne dans le canevas can1"
    global x1, y1, x2, y2, coul
    can1.create_line(x1,y1,x2,y2,width=2,fill=coul)

    # modification des coordonnées pour la ligne suivante :
    y2, y1 = y2+10, y1-10

def changecolor():
    "Changement aléatoire de la couleur du tracé"
    global coul
    pal=['purple','cyan','maroon','green','red','blue','orange','yellow']
    c = randrange(8)         # => génère un nombre aléatoire de 0 à 7
    coul = pal[c]


#------ Programme principal -------
from Tkinter import *
from random import randrange

# les variables suivantes seront utilisées de manière globale :
x1, y1, x2, y2 = 10, 190, 190, 10      # coordonnées de la ligne
coul = 'dark green'                    # couleur de la ligne

# Création du widget principal ("maître") :
fen1 = Tk()
# création des widgets "esclaves" :
can1 = Canvas(fen1,bg='dark grey',height=200,width=200)
can1.pack(side=LEFT)
```

31 Ces dessins pourront éventuellement être animés dans une phase ultérieure (voir plus loin)

```
bou1 = Button(fen1,text='Quitter',command=fen1.quit)
bou1.pack(side=BOTTOM)
bou2 = Button(fen1,text='Tracer une ligne',command=drawline)
bou2.pack()
bou3 = Button(fen1,text='Autre couleur',command=changecolor)
bou3.pack()

fen1.mainloop()         # démarrage du réceptionnaire d'événements

fen1.destroy()          # destruction (fermeture) de la fenêtre
```

Conformément à ce que nous avons expliqué dans le texte des pages précédentes, la fonctionnalité de ce programme est essentiellement assurée par les deux fonctions **drawline()** et **changecolor()**, qui seront activées par des événements, ceux-ci étant eux-mêmes définis dans la phase d'initialisation.

Dans cette phase d'initialisation, on commence par importer l'entièreté du module *Tkinter* ainsi qu'une fonction du module *random* qui permet de tirer des nombres au hasard. On crée ensuite les différents *widgets* par instanciation à partir des classes **Tk()**, **Canvas()** et **Button()**. (Remarquons au passage que le mot canevas s'écrit différemment en français et en anglais !)

L'initialisation se termine avec l'instruction **fen1.mainloop()** qui démarre le réceptionnaire d'événements. Les instructions qui suivent ne seront exécutées qu'à la sortie de cette boucle, sortie elle-même déclenchée par la méthode **fen1.quit()** (voir ci-après).

L'option **command** utilisée dans l'instruction d'instanciation des boutons permet de désigner la fonction qui devra être appelée lorsqu'un événement <*clic gauche de la souris sur le widget*> se produira. Il s'agit en fait d'un raccourci pour cet événement particulier, qui vous est proposé par *Tkinter* pour votre facilité parce que cet événement est celui que l'on associe naturellement à un widget de type bouton. Nous verrons plus loin qu'il existe d'autres techniques plus générales pour associer n'importe quel type d'événement à n'importe quel *widget*.

Les fonctions de ce script peuvent modifier les valeurs de certaines variables qui ont été définies au niveau principal du programme. Cela est rendu possible grâce à l'instruction **global** utilisée dans la définition de ces fonctions. Nous nous permettrons de procéder ainsi pendant quelque temps encore (ne serait-ce que pour vous habituer à distinguer les comportements des variables locales et globales), mais comme vous le comprendrez plus loin, cette pratique n'est pas tout à fait recommandable, surtout lorsqu'il s'agit d'écrire de grands programmes. Nous apprendrons une meilleure technique lorsque nous aborderons l'étude des classes (à partir de la page 145).

La commande liée au bouton "Quitter" appelle la méthode **quit()** de la fenêtre **fen1**. Cette méthode sert à fermer (quitter) le réceptionnaire d'événements (*mainloop*) associé à cette fenêtre. Lorsque cette méthode est activée, l'exécution du programme se poursuit avec les instructions qui suivent l'appel de *mainloop*. Dans notre exemple, cela consiste donc à effacer (*destroy*) la fenêtre.

Exercices : modifications au programme "Tracé de lignes" ci-dessus.

e 60. Comment faut-il modifier le programme pour ne plus avoir que des lignes de couleur cyan, maroon et green ?

e 61. Comment modifier le programme pour que toutes les lignes tracées soient horizontales et parallèles ?

e 62. Agrandissez le canevas de manière à lui donner une largeur de 500 unités et une hauteur de 650. Modifiez également la taille des lignes, afin que leurs extrémités se confondent avec les bords du canevas.

e 63. Ajoutez une fonction "drawline2" qui tracera deux lignes rouges en croix au centre du canevas : l'une horizontale et l'autre verticale. Ajoutez également un bouton portant l'indication "viseur". Un clic sur ce bouton devra provoquer l'affichage de la croix.

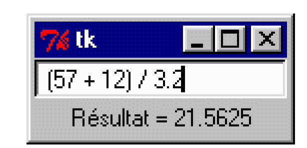
e 64. Reprenez le programme initial. Remplacez la méthode "create_line" par "create_rectangle". Que se passe-t-il ?
De la même façon, essayez aussi "create_arc", create_oval", et "create_polygon".
Pour chacune de ces méthodes, notez ce qu'indiquent les coordonnées fournies en paramètres.
(Remarque : pour le polygone, il est nécessaire de modifier légèrement le programme !)

e 65. - Supprimez la ligne "global x1, y1, x2, y2" dans la fonction "drawline" du programme original. Que se passe-t-il ? Pourquoi ?
- Si vous placez plutôt "x1, y1, x2, y2" entre les parenthèses, dans la ligne de définition de la fonction "drawline", de manière à transmettre ces variables à la fonction en tant que paramètres, le programme fonctionne-t-il encore ? (N'oubliez pas de modifier aussi la ligne du programme qui fait appel à cette fonction !)
- Si vous définissez "x1, y1, x2, y2 = 10, 390, 390, 10" à la place de "global x1, y1, ...", que se passe-t-il ? Pourquoi ? Quelle conclusion pouvez-vous tirer de tout cela ?

e 66. a) Créez un court programme qui dessinera les 5 anneaux olympiques dans un rectangle de fond blanc (white). Un boutton "Quitter" doit permettre de fermer la fenêtre.
b) Modifiez le programme ci-dessus en y ajoutant 5 boutons. Chacun de ces boutons provoquera le tracé de chacun des 5 anneaux

e 67. Dans votre cahier, établissez un tableau à deux colonnes. Vous y noterez à gauche les définitions des classes d'objets déjà rencontrées (avec leur liste de paramètres), et à droite les méthodes associées à ces classes (également avec leurs paramètres). Laisser de la place pour compléter ultérieurement.

8.3.2 Exemple graphique : calculatrice minimaliste



Bien que très court, le petit script ci-dessous implémente une calculatrice complète, avec laquelle vous pourrez même effectuer des calculs comportant des parenthèses et des fonctions scientifiques. N'y voyez rien d'extraordinaire. Toute cette fonctionnalité n'est qu'une conséquence du fait que vous utilisez un interpréteur plutôt qu'un compilateur pour exécuter vos programmes.

Comme vous le savez, le compilateur n'intervient qu'une seule fois, pour traduire l'ensemble de votre code source en un programme exécutable. Son rôle est donc terminé ***avant*** même l'exécution du programme. L'interpréteur, quant à lui, est toujours actif ***pendant*** l'exécution du programme, et donc tout à fait disponible pour traduire un nouveau code source quelconque, comme par exemple une expression mathématique entrée au clavier par l'utilisateur.

Les langages interprétés disposent donc toujours de fonctions permettant d'évaluer une chaîne de caractères comme une suite d'instructions du langage lui-même. Il devient alors possible de construire en peu de lignes des structures de programmes très dynamiques. Dans l'exemple ci-dessous, nous utilisons la fonction intégrée **eval()** pour analyser l'expression mathématique entrée par l'utilisateur dans le champ prévu à cet effet, et nous n'avons plus ensuite qu'à afficher le résultat.

```
# Exercice utilisant la bibliothèque graphique Tkinter et le module math

from Tkinter import *
from math import *

# définition de l'action à effectuer si l'utilisateur actionne
# la touche "enter" alors qu'il édite le champ d'entrée :

def evaluer(event):
    chaine.configure(text = "Résultat = " + str(eval(entree.get())))

# ----- Programme principal : -----

fenetre = Tk()
entree = Entry(fenetre)
entree.bind("<Return>", evaluer)
chaine = Label(fenetre)
entree.pack()
chaine.pack()

fenetre.mainloop()
```

Au début du script, nous commençons par importer les modules *Tkinter* et *math*, ce dernier étant nécessaire afin que la dite calculatrice puisse disposer de toutes les fonctions mathématiques et scientifiques usuelles : sinus, cosinus, racine carrée, etc.

Ensuite nous définissons une fonction **evaluer()**, qui sera en fait la commande exécutée par le programme lorsque l'utilisateur actionnera la touche *Return* (ou *Enter*) après avoir entré une expression mathématique quelconque dans le champ d'entrée décrit plus loin.

Cette fonction utilise la méthode **configure()** du widget **chaine**[32], pour modifier son attribut **text**. L'attribut en question reçoit donc ici une nouvelle valeur, déterminée par ce que nous avons écrit à la droite du signe égale : il s'agit en l'occurrence d'une chaîne de caractères construite dynamiquement, à l'aide de deux fonctions intégrées dans Python : **eval()** et **str()**, et d'une méthode associée à un widget *Tkinter* : la méthode **get()**.

eval() fait appel à l'interpréteur pour évaluer une expression Python qui lui est transmise dans une chaîne de caractères. Le résultat de l'évaluation est fourni en retour. Exemple :

```
chaine = "(25 + 8)/3"      # chaîne contenant une expression mathématique
res = eval(chaine)         # évaluation de l'expression contenue dans la chaîne
print res +5               # => le contenu de la variable res est numérique
```

str() transforme une expression numérique en chaîne de caractères. Nous devons faire appel à cette fonction parce que la précédente retourne une valeur numérique, que nous convertissons à nouveau en chaîne de caractères pour pouvoir l'incorporer au message "Résultat = ".

get() est une méthode associée aux *widgets* de la classe **Entry**. Dans notre petit programme exemple, nous utilisons un *widget* de ce type pour permettre à l'utilisateur d'**entrer** une expression numérique quelconque à l'aide de son clavier. La méthode **get()** permet en quelque sorte "d'extraire" du *widget* "entree" la chaîne de caractères qui lui a été fournie par l'utilisateur.

Le corps du programme principal contient la phase d'initialisation, qui se termine par la mise en route de l'observateur d'événements (*mainloop*). On y trouve l'instanciation d'une fenêtre **Tk()**, contenant un *widget* "chaine" instancié à partir de la classe **Label()**, et un *widget* "entree" instancié à partir de la classe **Entry()**.

Afin que ce dernier *widget* puisse vraiment faire son travail, c.à.d. transmettre au programme l'expression que l'utilisateur y aura encodée, ***nous lui associons un événement*** à l'aide de la méthode **bind()**[33] :

```
entree.bind("<Return>",evaluer)
```

Cette instruction signifie : "*Lier l'événement <pression sur la touche Return> à l'objet <entree>, le gestionnaire de cet événement étant la fonction <evaluer>*".

L'événement à prendre en charge est décrit dans une chaîne de caractères spécifique (dans notre exemple, il s'agit de la chaîne "<Return>"). Il existe un grand nombre de ces événements (mouvements et clics de la souris, enfoncement des touches du clavier, positionnement et redimensionnement des fenêtres, passage au premier plan, etc.). Vous trouverez la liste des chaînes spécifiques de tous ces événements dans les ouvrages de référence traitant de *Tkinter*.

Profitons de l'occasion pour observer encore une fois la syntaxe des instructions destinées à mettre en oeuvre une méthode associée à un objet :

objet.méthode(arguments)

On écrit d'abord le nom de l'objet sur lequel on désire intervenir, puis le point (qui fait office d'opérateur), puis le nom de la méthode à mettre en oeuvre ; entre les parenthèses associées à cette méthode, on indique enfin les arguments qu'on souhaite lui transmettre.

32 La méthode **configure()** peut s'appliquer à n'importe quel widget préexistant, pour en modifier les propriétés.

33 En anglais, le mot *bind* signifie "lier"

8.3.3 Exemple graphique : détection et positionnement d'un clic de souris

Dans la définition de la fonction "evaluer" de l'exemple précédent, vous aurez remarqué que nous avons fourni un argument **event** (entre les parenthèses).

Cet argument est obligatoire. Lorsque vous définissez une fonction gestionnaire d'événement qui est associée à un widget quelconque à l'aide de sa méthode **bind()**, vous devez toujours l'utiliser comme premier argument. Il s'agit d'un objet Python standard, créé automatiquement, qui permet de transmettre au gestionnaire d'événement un certain nombre d'attributs de cet événement :

- le type d'événement : déplacement de la souris, enfoncement ou relâchement de l'un de ses boutons, appui sur une touche du clavier, entrée du curseur dans une zone prédéfinie, ouverture ou fermeture d'une fenêtre, etc.
- une série de propriétés de l'événement : l'instant où il s'est produit, ses coordonnées, les caractéristiques du ou des widget(s) concerné(s), etc.

Nous n'allons pas entrer dans trop de détails. Si vous voulez bien encoder 2et expérimenter le petit script ci-dessous, vous aurez vite compris le principe.

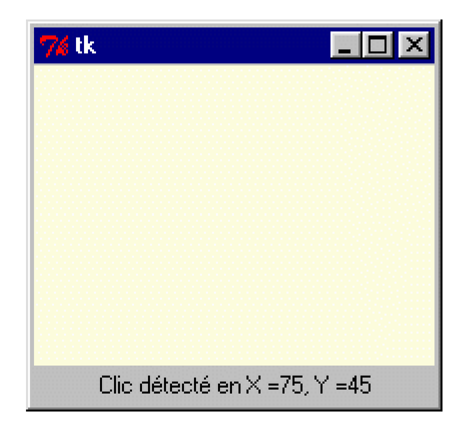
```
# Détection et positionnement d'un clic de souris dans une fenêtre :

from Tkinter import *

def pointeur(event):
  chaine['text'] = "Clic détecté en X =" + str(event.x) + ", Y =" + str(event.y)

fen = Tk()
cadre = Frame(fen, width =200, height =150, bg="light yellow")
cadre.bind("<Button-1>", pointeur)
cadre.pack()
chaine = Label(fen)
chaine.pack()

fen.mainloop()
```



Le script fait apparaître une fenêtre contenant un **cadre** (*frame*) rectangulaire de couleur jaune pâle, dans lequel l'utilisateur est invité à effectuer des clics de souris.

La méthode **bind()** du widget **cadre** associe l'événement <*clic à l'aide du premier bouton de la souris*> au gestionnaire d'événement "pointeur".

Ce gestionnaire d'événement peut utiliser les attributs **x** et **y** de l'objet **event** généré automatiquement par Python, pour construire la chaîne de caractères qui affichera la position de la souris au moment du clic.

Exercice :

e 68. Modifiez le script ci-dessus de manière à faire apparaître un petit cercle rouge à l'endroit où l'utilisateur a effectué son clic (vous devrez d'abord remplacer le widget Frame par un widget Canvas).

8.4 Les classes de widgets Tkinter

***Note** : Au long de ce cours, nous vous présenterons petit à petit le mode d'utilisation d'un certain nombre de widgets. Comprenez bien cependant qu'il n'entre pas dans nos intentions de fournir ici un manuel de référence complet sur Tkinter. Nous limiterons nos explications aux widgets qui nous semblent les plus intéressants d'un point de vue didactique, c.à.d. ceux qui pourront nous aider à mettre en évidence des concepts importants, tel le concept de classe. Veuillez donc consulter la littérature (voir page 8) si vous souhaitez davantage de précisions.*

Il existe 15 classes de base pour les widgets *Tkinter* :

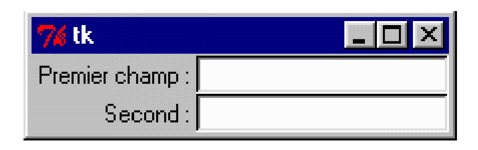
Widget	***Description***
Button	Un bouton classique, à utiliser pour provoquer l'exécution d'une commande quelconque.
Canvas	Un espace pour disposer divers éléments graphiques. Ce widget peut être utilisé pour dessiner, créer des éditeurs graphiques, et aussi pour implémenter des widgets personnalisés.
Checkbutton	Une "case à cocher" qui peut prendre deux états distincts (la case est cochée ou non). Un clic sur ce widget provoque le changement d'état.
Entry	Un champ d'entrée, dans lequel l'utilisateur du programme pourra insérer un texte quelconque à partir du clavier.
Frame	Une surface rectangulaire dans la fenêtre, où l'on peut disposer d'autres widgets. Cette surface peut être colorée. Elle peut aussi être décorée d'une bordure.
Label	Un texte (ou libellé) quelconque (éventuellement une image).
Listbox	Une liste de choix proposés à l'utilisateur, généralement présentés dans une sorte de boîte. On peut également configurer la Listbox de telle manière qu'elle se comporte comme une série de "boutons radio" ou de cases à cocher.
Menu	Un menu. Ce peut être un menu déroulant attaché à la barre de titre, ou bien un menu "pop up" apparaissant n'importe où à la suite d'un clic.
Menubutton	Un bouton-menu, à utiliser pour implémenter des menus déroulants.
Message	Permet d'afficher un texte. Ce widget est une variante du widget Label, qui permet d'adapter automatiquement le texte affiché à une certaine taille ou à un certain rapport largeur/hauteur.
Radiobutton	Représente (par un point noir dans un petit cercle) une des valeurs d'une variable qui peut en posséder plusieurs. Cliquer sur un "bouton radio" donne la valeur correspondante à la variable, et "vide" tous les autres boutons radio associés à la même variable.
Scale	Vous permet de faire varier de manière très visuelle la valeur d'une variable, en déplaçant un curseur le long d'une règle.
Scrollbar	"ascenseur" ou "barre de défilement" que vous pouvez utiliser en association avec les autres widgets : Canvas, Entry, Listbox, Text.
Text	Affichage de texte formatté. Permet aussi à l'utilisateur d'éditer le texte affiché. Des images peuvent également être insérées.
Toplevel	Une fenêtre affichée séparément, "par-dessus".

Ces classes de *widgets* intègrent chacune un grand nombre de méthodes. On peut aussi leur associer (lier) des événements, comme nous venons de le voir dans les pages précédentes. Vous allez apprendre en outre que tous ces widgets peuvent être positionnés dans les fenêtres à l'aide de trois méthodes différentes : la méthode **grid()**, la méthode **pack()** et la méthode **place()**.

L'utilité de ces méthodes apparaît clairement lorsque l'on s'efforce de réaliser des programmes ***portables*** (c.à.d. susceptibles de fonctionner indifféremment sur des systèmes d'exploitation aussi différents que UN!X, M@C ou Window$), et dont les fenêtres soient redimensionnables.

8.5 Utilisation de la méthode grid() pour contrôler la disposition des widgets

Jusqu'à présent, nous avons toujours disposé les widgets dans leur fenêtre, à l'aide de la méthode **pack()**. Cette méthode présentait l'avantage d'être extraordinairement simple, mais elle ne nous donnait pas beaucoup de liberté pour disposer les widgets à notre guise. Comment faire, par exemple, pour obtenir la fenêtre ci-contre ?



Nous pourrions effectuer un certain nombre de tentatives en fournissant à la méthode **pack()** des arguments de type "side =", comme nous l'avons déjà fait précédemment, mais cela ne nous mène pas très loin. Essayons par exemple :

```
from Tkinter import *

fen1 = Tk()
txt1 = Label(fen1, text = 'Premier champ :')
txt2 = Label(fen1, text = 'Second :')
entr1 = Entry(fen1)
entr2 = Entry(fen1)
txt1.pack(side =LEFT)
txt2.pack(side =LEFT)
entr1.pack(side =RIGHT)
entr2.pack(side =RIGHT)

fen1.mainloop()
```

... mais le résultat n'est pas vraiment celui que nous recherchions !!! :



Pour mieux comprendre comment fonctionne la méthode **pack()**, vous pouvez encore essayer différentes combinaisons d'options, telles que **side =TOP, side =BOTTOM**, pour chacun de ces quatre widgets. Mais vous n'arriverez certainement pas à obtenir ce qui vous a été demandé. Vous pourriez peut-être y parvenir en définissant deux widgets **Frame()** supplémentaires, et en y incorporant ensuite séparément les widgets **Label()** et **Entry()**. Cela devient fort compliqué.

Il est temps que nous apprenions à utiliser une autre approche du problème. Veuillez donc analyser le script ci-dessous : il contient en effet (presque) la solution :

```
from Tkinter import *

fen1 = Tk()
txt1 = Label(fen1, text = 'Premier champ :')
txt2 = Label(fen1, text = 'Second :')
entr1 = Entry(fen1)
entr2 = Entry(fen1)
txt1.grid(row =0)
txt2.grid(row =1)
entr1.grid(row =0, column =1)
entr2.grid(row =1, column =1)
fen1.mainloop()
```

Dans ce script, nous avons donc remplacé la méthode **pack()** par la méthode **grid()**. Comme vous pouvez le constater, l'utilisation de la méthode **grid()** est très simple. Cette méthode considère la fenêtre comme un tableau (ou une grille). Il suffit alors de lui indiquer dans quelle ligne (*row*) et dans quelle colonne (*column*) de ce tableau on souhaite placer les *widgets*. On peut numéroter les lignes et les colonnes comme on veut, en partant de zéro, ou de un, ou encore d'un nombre quelconque : *Tkinter* ignorera les lignes et colonnes vides. Notez cependant que si vous ne fournissez aucun numéro pour une ligne ou une colonne, la valeur par défaut sera zéro.

Tkinter détermine automatiquement le nombre de lignes et de colonnes nécessaire. Mais ce n'est pas tout : si vous examinez en détail la petite fenêtre produite par le script ci-dessus, vous constaterez que nous n'avons pas encore tout à fait atteint le but poursuivi. Les deux chaînes apparaissant dans la partie gauche de la fenêtre sont centrées, alors que nous souhaitions les aligner l'une et l'autre par la droite. Pour obtenir ce résultat, il nous suffit d'ajouter un argument dans l'appel de la méthode **grid()** utilisée pour ces *widgets*. L'option **sticky** peut prendre l'une des quatre valeurs **N, S, W, E** (les quatre points cardinaux en anglais). En fonction de cette valeur, on obtiendra un alignement des *widgets* par le haut, par le bas, par la gauche ou par la droite. Remplacez donc les deux premières instructions **grid()** du script par :

```
txt1.grid(row =0, sticky =E)
txt2.grid(row =1, sticky =E)
```

... et vous atteindrez enfin exactement le but recherché.

Analysons à présent la fenêtre suivante :



Cette fenêtre comporte 3 colonnes : une première avec les 3 chaînes de caractères, une seconde avec les 3 champs d'entrée, et une troisième avec l'image. Les deux premières colonnes comportent chacune 3 lignes, mais l'image située dans la dernière colonne "s'étale" en quelque sorte sur les trois.

Le code correspondant est le suivant :

```
from Tkinter import *

fen1 = Tk()

# création de widgets 'Label' et 'Entry' :
txt1 = Label(fen1, text ='Premier champ :')
txt2 = Label(fen1, text ='Second :')
txt3 = Label(fen1, text ='Troisième :')
entr1 = Entry(fen1)
entr2 = Entry(fen1)
entr3 = Entry(fen1)

# création d'un widget 'Canvas' contenant une image bitmap :
can1 = Canvas(fen1, width =160, height =160, bg ='white')
photo = PhotoImage(file ='Martin_P.gif')
item = can1.create_image(80, 80, image =photo)

# Mise en page à l'aide de la méthode 'grid' :
txt1.grid(row =1, sticky =E)
txt2.grid(row =2, sticky =E)
txt3.grid(row =3, sticky =E)
entr1.grid(row =1, column =2)
entr2.grid(row =2, column =2)
entr3.grid(row =3, column =2)
can1.grid(row =1, column =3, rowspan =3, padx =10, pady =5)

# démarrage :
fen1.mainloop()
```

Pour pouvoir faire fonctionner ce script, il vous faudra probablement remplacer le nom du fichier image (Martin_P.gif) par le nom d'une image de votre choix. Attention : la bibliothèque *Tkinter* standard n'accepte qu'un petit nombre de formats pour cette image. Choisissez de préférence le format GIF.

Nous pouvons remarquer un certain nombre de choses dans ce script :

1. La technique utilisée pour incorporer une image :
   Tkinter ne permet pas d'insérer directement une image dans une fenêtre. Il faut d'abord installer un canevas, et ensuite positionner l'image dans celui-ci. Nous avons opté pour un canevas de couleur blanche, afin de pouvoir le distinguer de la fenêtre. Vous pouvez remplacer le paramètre **bg ='white'** par **bg ='gray'** si vous souhaitez que le canevas devienne invisible. Étant donné qu'il existe de nombreux types d'images, nous devons en outre déclarer l'objet image comme étant un *bitmap* GIF, à partir de la classe **PhotoImage()**[34].

2. La ligne où nous installons l'image dans le canevas est la ligne :
   ```
   item = can1.create_image(80, 80, image =photo)
   ```
   Pour employer un vocabulaire correct, nous dirons que nous utilisons ici la ***méthode*** **create_image()** associée à ***l'objet*** **can1** (lequel objet est lui-même une ***instance*** de la ***classe*** **Canvas**). Les deux premiers ***arguments*** transmis (**80, 80**) indiquent les coordonnées x et y du canevas où il faut placer le centre de l'image. (Les dimensions du canevas étant de 160x160, notre choix aboutira donc à un centrage de l'image au milieu du canevas).

3. La numérotation des lignes et colonnes dans la méthode **grid()** :
   On peut constater que la numérotation des lignes et des colonnes dans la méthode **grid()** utilisée ici commence cette fois à partir de 1 (et non à partir de zéro comme dans le script précédent). Comme nous l'avons déjà signalé plus haut, ce choix de numérotation est tout à fait libre.
   On pourrait tout aussi bien numéroter : 5, 10, 15, 20... puisque *Tkinter* ignore les lignes et les colonnes vides. Numéroter à partir de l augmente probablement la lisibilité de notre code.

4. Les arguments utilisés avec **grid()** pour positionner le canevas :
   ```
   can1.grid(row =1, column =3, rowspan =3, padx =10, pady =5)
   ```
   Les deux premiers arguments indiquent que le canevas sera placé dans la première ligne de la troisième colonne. Le troisième (**rowspan =3**) indique qu'il pourra "s'étaler" sur trois lignes.
   Les deux derniers (**padx =10**, **pady =5**) indiquent la dimension de l'espace qu'il faut réserver autour de ce widget (en largeur et en hauteur).

5. Et tant que nous y sommes, profitons de cet exemple de script que nous avons déjà bien décortiqué, pour apprendre à simplifier quelque peu notre code ...

34 Il existe d'autres classes d'images, mais pour les utiliser il faut importer dans le script d'autres modules graphiques que la seule bibliothèque *Tkinter*. Vous pouvez par exemple expérimenter la bibliothèque PIL (*Python Imaging Library*).

8.6 Composition d'instructions pour écrire un code plus compact

Du fait que Python est un langage de programmation de haut niveau, il est souvent possible (et souhaitable) de retravailler un script afin de le rendre plus compact.

Vous pouvez par exemple assez fréquemment utiliser la composition d'instructions pour appliquer la méthode de mise en page des widgets (**grid()**, **pack()** ou **place()**) au moment même où vous créez ces widgets. Le code correspondant devient alors un peu plus simple, et parfois plus lisible. Vous pouvez par exemple remplacer les deux lignes :

```
txt1 = Label(fen1, text ='Premier champ :')
txt1.grid(row =1, sticky =E)
```

du script précédent par une seule, telle que :

```
Label(fen1, text ='Premier champ :').grid(row =1, sticky =E)
```

Dans cette nouvelle écriture, vous pouvez constater que nous faisons l'économie de la variable intermédiaire **txt1**. Nous avions utilisé cette variable pour bien dégager les étapes successives de notre démarche, mais elle n'est pas toujours indispensable. Le simple fait d'invoquer la classe **Label()** provoque en effet l'instanciation d'un objet de cette classe, même si l'on ne mémorise pas la référence de cet objet dans une variable (*Tkinter* la conserve de toute façon dans sa représentation interne de la fenêtre). Si l'on procède ainsi, la référence est perdue pour le restant du script, mais elle peut tout de même être transmise à une méthode de mise en page telle que **grid()** au moment même de l'instanciation, en une seule instruction composée. Voyons cela un peu plus en détail :

Jusqu'à présent, nous avons créé des objets divers (par instanciation à partir d'une classe quelconque), en les affectant à chaque fois à des variables. Par exemple, lorsque nous avons écrit :

```
txt1 = Label(fen1, text ='Premier champ :')
```

nous avons créé une instance de la classe **Label()**, que nous avons assignée à la variable `txt1`.

La variable **txt1** peut alors être utilisée pour faire référence à cette instance, partout ailleurs dans le script, mais dans les faits nous ne l'utilisons qu'une seule fois pour lui appliquer la méthode **grid()**, le widget dont il est question n'étant rien d'autre qu'une simple étiquette descriptive. Or, créer ainsi une nouvelle variable pour n'y faire référence ensuite qu'une seule fois (et directement après sa création) n'est pas une pratique très recommandable, puisqu'elle consiste à réserver inutilement un certain espace mémoire.

Lorsque ce genre de situation se présente, il est plus judicieux d'utiliser la composition d'instructions. Par exemple, on préférera le plus souvent remplacer les deux instructions :

```
somme = 45 + 72
print somme
```

par une seule instruction composée, telle que :

```
print 45 + 72
```

on fait ainsi l'économie d'une variable.

De la même manière, lorsque l'on met en place des widgets auxquels on ne souhaite plus revenir par après, comme c'est souvent le cas pour les widgets de la classe **Label()**, on peut en général appliquer la méthode de mise en page (**grid()** , **pack()** ou **place()**) directement au moment de la création du widget, en une seule instruction composée.

Cela s'applique seulement aux widgets qui ne sont plus référencés après qu'on les ait créés. ***Tous les autres doivent impérativement être assignés à des variables***, afin que l'on puisse encore interagir avec eux ailleurs dans le script.

Et dans ce cas, il faut obligatoirement utiliser deux instructions distinctes, l'une pour instancier le widget et l'autre pour lui appliquer ensuite la méthode de mise en page. Vous ne pouvez pas, par exemple, construire une instruction composée telle que :

```
entree = Entry(fen1).pack()            # faute de programmation !!!
```

En apparence, cette instruction devrait instancier un nouveau widget et l'assigner à la variable **entree**, la mise en page s'effectuant dans la même opération à l'aide de la méthode **pack()**.
Dans la réalité, cette instruction produit bel et bien un nouveau widget de la classe **Entry()**, et la méthode **pack()** effectue bel et bien sa mise en page dans la fenêtre, mais la valeur qui est mémorisée dans la variable **entree** est la valeur de retour de la méthode **pack()** : ***ce n'est pas la référence du widget***. Et vous ne pouvez rien en faire (il s'agit d'une valeur nulle).

Pour obtenir une vraie référence du widget, vous devez utiliser deux instructions :

```
entree = Entry(fen1)                   # instanciation du widget
entree.pack()                          # application de la mise en page
```

Note : Lorsque vous utilisez la méthode **grid()**, vous pouvez simplifier encore un peu votre code, en omettant l'indication de nombreux numéros de lignes et de colonnes. A partir du moment où c'est la la méthode **grid()** qui est utilisée pour positionner les widgets, *Tkinter* considère en effet qu'il existe forcément des lignes et des colonnes[35]. Si un numéro de ligne ou de colonne n'est pas indiqué, le widget correspondant est placé dans la première case ***vide*** disponible.

Le script ci-dessous intègre les simplifications que nous venons d'expliquer :

```
from Tkinter import *
fen1 = Tk()

# création de widgets Label(), Entry(), et Check() :
Label(fen1, text = 'Premier champ :').grid(sticky =E)
Label(fen1, text = 'Second :').grid(sticky =E)
Label(fen1, text = 'TroisiÃ¨me :').grid(sticky =E)
entr1 = Entry(fen1)
entr2 = Entry(fen1)                  # ces widgets devront certainement
entr3 = Entry(fen1)                  # être  référencés plus loin :
entr1.grid(row =0, column =1)        # il faut donc les assigner chacun
entr2.grid(row =1, column =1)        # à une variable distincte
entr3.grid(row =2, column =1)
chek1 = Checkbutton(fen1, text ='Case à cocher, pour voir')
chek1.grid(columnspan =2)

# création d'un widget 'Canvas' contenant une image bitmap :
can1 = Canvas(fen1, width =160, height =160, bg ='white')
photo = PhotoImage(file ='Martin_P.gif')
can1.create_image(80,80, image =photo)
can1.grid(row =0, column =2, rowspan =4, padx =10, pady =5)

# démarrage :
fen1.mainloop()
```

35 Surtout, n'utilisez pas plusieurs méthodes de positionnement différentes dans la même fenêtre !
Les méthodes grid(), pack() et place() sont mutuellement exclusives.

8.7 Modification des propriétés d'un objet - Animation

A ce stade de votre apprentissage, vous souhaitez certainement pouvoir faire apparaître un petit dessin quelconque dans un canevas, et puis le déplacer à volonté, par exemple à l'aide de boutons. Veuillez donc écrire, tester, puis analyser le script ci-dessous :

```
from Tkinter import *

# procédure générale de déplacement :
def avance(gd, hb):
    global x1, y1
    x1, y1 = x1 +gd, y1 +hb
    can1.coords(oval1, x1, y1, x1+30, y1+30)

# gestionnaires d'événements :
def depl_gauche():
    avance(-10, 0)

def depl_droite():
    avance(10, 0)

def depl_haut():
    avance(0, -10)

def depl_bas():
    avance(0, 10)

#------ Programme principal -------

# les variables suivantes seront utilisées de manière globale :
x1, y1 = 10, 10         # coordonnées initiales

# Création du widget principal ("maître") :
fen1 = Tk()
fen1.title("Exercice d'animation avec Tkinter")

# création des widgets "esclaves" :
can1 = Canvas(fen1,bg='dark grey',height=300,width=300)
oval1 = can1.create_oval(x1,y1,x1+30,y1+30,width=2,fill='red')
can1.pack(side=LEFT)
Button(fen1,text='Quitter',command=fen1.quit).pack(side=BOTTOM)
Button(fen1,text='Gauche',command=depl_gauche).pack()
Button(fen1,text='Droite',command=depl_droite).pack()
Button(fen1,text='Haut',command=depl_haut).pack()
Button(fen1,text='Bas',command=depl_bas).pack()

# démarrage du réceptionnaire d'évènements (boucle principale) :
fen1.mainloop()
```

Le corps de ce programme reprend de nombreuses éléments connus : nous y créons une fenêtre **fen1**, dans laquelle nous installons un canevas contenant lui-même un cercle coloré, plus cinq boutons de contrôle. Veuillez remarquer au passage que nous n'instancions pas les *widgets* "boutons" dans des variables (c'est inutile, puisque nous n'y faisons plus référence par après) : nous devons donc appliquer la méthode **pack()** directement au moment de la création de ces objets.

La vraie nouveauté de ce programme réside dans la fonction **avance()** définie au début du script. Chaque fois qu'elle sera appelée, cette fonction redéfinira les coordonnées de l'objet "cercle coloré" que nous avons installé dans le canevas, ce qui provoquera l'animation de cet objet.

Cette manière de procéder est tout à fait caractéristique de la programmation "orientée objet" :
On commence par créer des objets, et puis ***on agit sur ces objets en modifiant leurs propriétés***, par l'intermédiaire de méthodes.

En programmation "traditionnelle" (c.à.d. sans utilisation d'objets), on anime des figures en les effaçant à un endroit pour les redessiner ensuite un petit peu plus loin. En programmation "orientée objet", par contre, ces tâches sont prises en charge automatiquement par les classes dont les objets dérivent, et il ne faut donc pas perdre son temps à les reprogrammer.

Exercices :

e 69. Ecrivez un programme qui fait apparaître une fenêtre avec un canevas. Dans ce canevas on verra deux cercles (de tailles et de couleurs différentes), qui sont censés représenter deux astres. Des boutons doivent permettre de les déplacer à volonté tous les deux dans toutes les directions. Sous le canevas, le programme doit afficher en permanence : a) la distance séparant les deux astres; b) la force gravitationnelle qu'ils exercent l'un sur l'autre (Penser à afficher en haut de fenêtre les masses choisies pour chacun d'eux, ainsi que l'échelle des distances). Dans cet exercice, vous utiliserez évidemment la loi de la gravitation universelle de Newton (cfr. exercice 42, page 60, et votre cours de Physique générale).

e 70. En vous inspirant du programme qui détecte les clics de souris dans un canevas, modifiez le programme ci-dessus pour y réduire le nombre de boutons : pour déplacer un astre, il suffira de le choisir avec un bouton, et ensuite de cliquer sur le canevas pour que cet astre se positionne à l'endroit où l'on a cliqué.

e 71. Extension du programme ci-dessus. Faire apparaître un troisième astre, et afficher en permanence la force résultante agissant sur chacun des trois (en effet : chacun subit en permanence l'attraction gravitationnelle exercée par les deux autres !).

e 72. Même exercice avec des charges électriques (loi de Coulomb). Donner cette fois une possibilité de choisir le signe des charges.

e 73. Ecrivez un petit programme qui fait apparaître une fenêtre avec deux champs : l'un indique une température en degrés *Celsius*, et l'autre la même température exprimée en degrés *Fahrenheit*. Chaque fois que l'on change une quelconque des deux températures, l'autre est corrigée en conséquence. Pour convertir les degrés *Fahrenheit* en *Celsius* et vice-versa, on utilise la formule $T_F = T_C \times 1{,}80 + 32$ (cfr. cours de Physique générale). Revoyez aussi le petit programme concernant la calculatrice simplifiée.

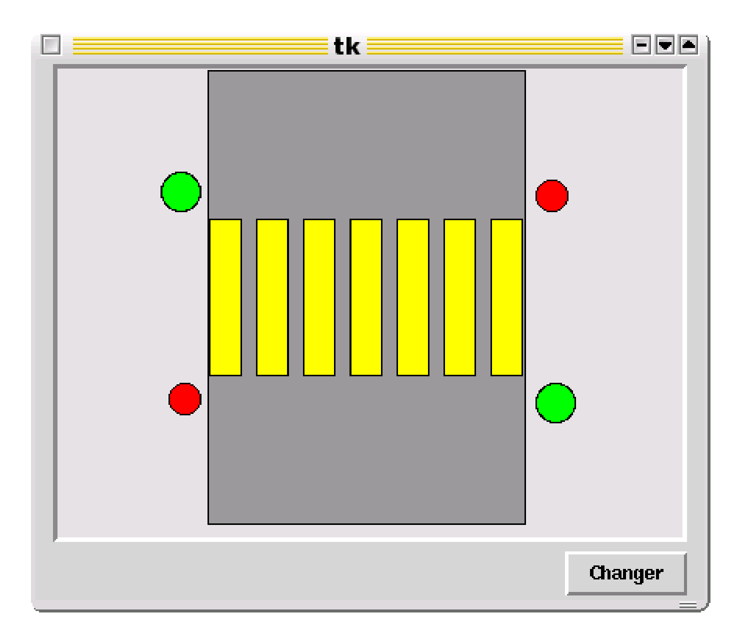
e 74. Écrivez un programme qui fasse apparaître une fenêtre avec un canevas. Dans ce canevas, placez un petit cercle censé représenter une balle. Sous le canevas, placez un bouton. Chaque fois que l'on clique sur le bouton, la balle doit avancer d'une petite distance vers la droite, jusqu'à ce qu'elle atteigne l'extrémité du canevas. Si l'on continue à cliquer, la balle doit alors revenir en arrière jusqu'à l'autre extrémité, et ainsi de suite.

e 75. Améliorez le programme ci-dessus pour que la balle décrive cette fois une trajectoire circulaire ou elliptique dans le canevas (lorsque l'on clique continuellement). Note : pour arriver au résultat escompté, vous devrez nécessairement définir une variable qui représentera l'angle décrit, et utiliser les fonctions sinus et cosinus pour positionner la balle en fonction de cet angle.

e 76. Modifiez le programme ci-dessus, de telle manière que la balle en se déplaçant laisse derrière elle une trace de la trajectoire décrite.

e 77. Modifiez le programme ci-dessus de manière à tracer d'autres figures. Consultez votre professeur pour des suggestions (courbes de Lissajous).

e 78. Ecrivez un programme qui fasse apparaître une fenêtre avec un canevas et un bouton. Dans le canevas, tracez un rectangle gris foncé, lequel représentera une route, et par-dessus, une série de rectangles jaunes censés représenter un passage pour piétons. Ajoutez quatre cercles colorés pour figurer les feux de circulation concernant les piétons et les véhicules. Chaque utilisation du bouton devra provoquer le changement de couleur des feux :



e 79. Écrivez un programme qui montre un canevas dans lequel est dessiné un circuit électrique simple (générateur + interrupteur + résistance). La fenêtre doit être pourvue de champs d'entrée qui permettront de paramétrer chaque élément (c.à.d. choisir les valeurs des résistances et tensions). L'interrupteur doit être fonctionnel (Prévoyez un bouton "Marche/arrêt" pour cela). Des "étiquettes" doivent afficher en permanence les tensions et intensités résultant des choix opérés par l'utilisateur.

8.8 Animation automatique - Récursivité

Pour conclure cette première prise de contact avec l'interface graphique *Tkinter*, voici un dernier exemple d'animation, qui fonctionne cette fois de manière autonome dès qu'on l'a mise en marche.

```
from Tkinter import *

def move():
    "déplacement de la balle"
    global x1, y1, dx, dy, flag
    x1, y1 = x1 +dx, y1 + dy
    if x1 >360:
        x1, dx, dy = 360, 0, 15
    if y1 >360:
        y1, dx, dy = 360, -15, 0
    if x1 <10:
        x1, dx, dy = 10, 0, -15
    if y1 <10:
        y1, dx, dy = 10, 15, 0
    can1.coords(oval1,x1,y1,x1+30,y1+30)
    if flag >0:
        fen1.after(50, move)          # boucler après 50 millisecondes

def stop_it():
    "arrêt de l'animation"
    global flag
    flag =0

def start_it():
    "démarrage de l'animation"
    global flag
    flag = flag +1      # préférable à flag =1 :
    if flag ==1:        # voir explications dans le texte
        move()

#========== Programme principal =============
# les variables suivantes seront utilisées de manière globale :
x1, y1 = 10, 10         # coordonnées initiales
dx, dy = 15, 0          # 'pas' du déplacement
flag =0                 # commutateur

# Création du widget principal ("maître") :
fen1 = Tk()
fen1.title("Exercice d'animation avec Tkinter")

# création des widgets "esclaves" (canevas + balle, boutons):
can1 = Canvas(fen1,bg='dark grey',height=400,width=400)
can1.pack(side=LEFT)
oval1 = can1.create_oval(x1,y1,x1+30,y1+30,width=2,fill='red')
Button(fen1,text='Quitter',command=fen1.quit).pack(side=BOTTOM)
Button(fen1,text='Démarrer',command=start_it).pack()
Button(fen1,text='Arrêter',command=stop_it).pack()

# démarrage du réceptionnaire d'évènements (boucle principale) :
fen1.mainloop()
```

La seule nouveauté mise en oeuvre dans ce script se trouve tout à la fin de la définition de la fonction **move()** : vous y noterez l'utilisation de la méthode **after()**. Cette méthode peut s'appliquer à une fenêtre quelconque. Elle déclenche l'appel d'une fonction ***après qu'un certain laps de temps se soit écoulé***. Ainsi par exemple, **window.after(200,qqc)** déclenche pour le *widget* **window** un appel de la fonction **qqc()** après une pause de 200 millisecondes.

Dans notre script, la fonction qui est appelée par la méthode **after()** est la fonction **move()** elle-même. Nous utilisons donc ici pour la première fois une technique de programmation très puissante, que l'on appelle ***récursivité***. Pour faire simple, nous dirons que ***la récursivité est ce qui se passe lorsqu'une fonction s'appelle elle-même***. On obtient bien évidemment ainsi un bouclage, qui peut se perpétuer indéfiniment si l'on ne prévoit pas aussi un moyen pour l'interrompre.

Voyons comment cela fonctionne dans notre exemple :

La fonction **move()** est invoquée une première fois lorsque l'on clique sur le bouton "Démarrer". Elle effectue son travail (c.à.d. positionner la balle), puis elle s'invoque elle-même après une petite pause. Elle repart donc pour un second tour, puis s'invoque elle-même à nouveau, et ainsi de suite indéfiniment...

C'est du moins ce qui se passerait si nous n'avions pas pris la précaution de placer quelque part dans la boucle une instruction de sortie. En l'occurrence, il s'agit d'un simple test conditionnel : à chaque itération de la boucle, nous examinons le contenu de la variable **flag** à l'aide d'une instruction **if**. Si le contenu de la variable **flag** est zéro, alors le bouclage ne s'effectue plus et l'animation s'arrête. **flag** étant une variable globale, nous pouvons aisément changer sa valeur à l'aide d'autres fonctions, celles que nous avons associées aux boutons "Démarrer" et "Arrêter".

Nous obtenons ainsi un mécanisme simple pour lancer ou arrêter notre animation :

Un premier clic sur le bouton "Démarrer" assigne une valeur non-nulle à la variable **flag**, puis provoque immédiatement un premier appel de la fonction **move()**. Celle-ci s'exécute et continue ensuite à s'appeler elle-même toutes les 50 millisecondes, tant que **flag** ne revient pas à zéro. Si l'on continue à cliquer sur le bouton "Démarrer", la valeur de **flag** augmente mais rien d'autre ne se passe, parce que **move()** n'est appelée que lorsque **flag** vaut 1. On évite ainsi le démarrage de plusieurs boucles concurrentes.

Le bouton "Arrêter" remet **flag** à zéro et la boucle s'interrompt.

Exercices :

e 80. Dans la fonction **start_it()**, remplacez l'instruction **flag = flag +1** par **flag =1**.
Que se passe-t-il ?
(Cliquez plusieurs fois sur le bouton "Démarrer") Tâchez d'exprimer le plus clairement possible votre explication des faits observés.

e 81. Modifiez le programme de telle façon que la balle change de couleur à chaque "virage".

e 82. Modifiez le programme de telle façon que la balle effectue des mouvements obliques comme une bille de billard qui rebondit sur les bandes ("en zig-zag").

e 83. Modifiez le programme de manière à obtenir d'autres mouvements. Tâchez par exemple d'obtenir un mouvement circulaire. (Comme dans les exercices de la page 100).

e 84. Modifiez ce programme, ou bien écrivez-en un autre similaire, de manière à simuler le mouvement d'une balle qui tombe (sous l'effet de la pesanteur), et rebondit sur le sol. Attention : il s'agit cette fois de mouvements accélérés !

e 85. A partir des scripts précédents, vous pouvez à présent écrire un programme de jeu fonctionnant de la manière suivante :
Une balle se déplace au hasard sur un canevas, à vitesse faible. Le joueur doit essayer de cliquer sur cette balle à l'aide de la souris. S'il y arrive, il gagne un point mais la balle se déplace désormais un peu plus vite, et ainsi de suite. Arrêter le jeu après un certain nombre de clics et afficher le score atteint.

e 86. Variante du jeu précédent : chaque fois que le joueur parvient à l'"attraper", la balle devient plus petite (elle peut également changer de couleur).

e 87. Écrivez un programme dans lequel évoluent plusieurs balles de couleurs différentes, qui rebondissent les unes sur les autres ainsi que sur les parois.

e 88. Perfectionnez le jeu des précédents exercices en y intégrant l'algorithme ci-dessus. Il s'agit à présent pour le joueur de cliquer seulement sur la balle rouge. Un clic erroné (sur une balle d'une autre couleur) lui fait perdre des points.

e 89. Écrivez un programme qui simule le mouvement de 2 planètes tournant autour du soleil sur des orbites circulaires différentes (ou deux électrons tournant autour d'un noyau d'atome...).

e 90. Écrivez un programme pour le jeu du serpent : un "serpent" (constitué en faite d'une courte ligne de carrés) se déplace sur le canevas dans l'une des 4 directions : droite, gauche, haut, bas. Le joueur peut à tout moment changer la direction suivie par le serpent à l'aide des touches fléchées du clavier. Sur le canevas se trouvent également des "proies" (des petits cercles fixes disposés au hasard). Il faut diriger le serpent de manière à ce qu'il "mange" les proies sans arriver en contact avec les bords du canevas. A chaque fois qu'une proie est mangée, le serpent s'allonge d'un carré, le joueur gagne un point, et une nouvelle proie apparaît ailleurs. La partie s'arrête lorsque le serpent touche l'une des parois, ou lorsqu'il a atteint une certaine taille.

e 91. Perfectionnement du jeu précédent : la partie s'arrête également si le serpent se "recoupe".

Chapitre 9 : Les fichiers

9.1 Utilité des fichiers

Jusqu'à présent, les programmes que nous avons réalisés ne traitaient qu'un très petit nombre de données. Nous pouvions donc à chaque fois inclure ces données dans le corps du programme lui-même (par exemple dans une liste). Cette façon de procéder devient cependant tout à fait inadéquate lorsque l'on souhaite traiter une quantité d'information plus importante.

Imaginons par exemple que nous voulons écrire un petit programme exerciseur qui fasse apparaître à l'écran des questions à choix multiple, avec traitement automatique des réponses de l'utilisateur. Comment allons-nous mémoriser le texte des questions elles-mêmes ?

L'idée la plus simple consiste à placer chacun de ces textes dans une variable, en début de programme, avec des instructions d'affectation du genre :

```
a = "Quelle est la capitale du Guatémala ?"
b = "Qui à succédé à Henri IV ?"
c = "Combien font 26 x 43 ?"
     ... etc.
```

Cette idée est malheureusement beaucoup trop simpliste. Tout va se compliquer en effet lorsque nous essayerons d'élaborer la suite du programme, c.à.d. les instructions qui devront servir à sélectionner au hasard l'une ou l'autre de ces questions pour les présenter à l'utilisateur. Employer par exemple une longue suite d'instructions **if ... elif ... elif ...** comme dans l'exemple ci-dessous n'est certainement pas la bonne solution (ce serait d'ailleurs bien pénible à écrire : n'oubliez pas que nous souhaitons traiter un grand nombre de questions !) :

```
if choix == 1:
    selection = a
elif choix == 2:
    selection = b
elif choix == 3:
    selection = c
     ... etc.
```

La situation se présente déjà beaucoup mieux si nous faisons appel à une liste :

```
liste = ["Qui a vaincu Napoléon à Waterloo ?",
         "Comment traduit-on 'informatique' en anglais ?",
         "Quelle est la formule chimique du méthane ?", ... etc ...]
```

On peut en effet extraire n'importe quel élément de cette liste à l'aide de son indice. Exemple :

```
print liste[2]        ===>  "Quelle est la formule chimique du méthane ?"
```

(rappel : l'indiçage commence à partir de zéro)

Même si cette façon de procéder est déjà nettement meilleure que la précédente, nous sommes toujours confrontés à plusieurs problèmes gênants :

- La lisibilité du programme va se détériorer très vite lorsque le nombre de questions deviendra important. En corollaire, nous accroîtrons la probabilité d'insérer l'une ou l'autre erreur de syntaxe dans la définition de cette liste "kilométrique". De telles erreurs seront bien difficiles à débusquer.
- L'ajout de nouvelles questions, ou la modification de certaines d'entre elles, imposeront à chaque fois de rouvrir le code source du programme. En corollaire, il deviendra malaisé de retravailler ce même code source, puisqu'il comportera de nombreuses lignes de données encombrantes.
- L'échange de données avec d'autres programmes (peut-être écrits dans d'autres langages) est tout simplement impossible, puisque ces données font partie du programme lui-même.

Cette dernière remarque nous suggère la direction à prendre : il est temps que nous apprenions à ***séparer les données, et les programmes qui les traitent, dans des fichiers différents***.

Pour que cela devienne possible, nous devrons doter nos programmes de divers mécanismes permettant de créer des fichiers, d'y envoyer des données et de les récupérer par après.

Les langages de programmation proposent des jeux d'instructions plus ou moins sophistiqués pour effectuer ces tâches. Lorsque les quantités de données deviennent très importantes, il devient d'ailleurs rapidement nécessaire de structurer les relations entre ces données, et l'on doit alors élaborer des systèmes appelés ***bases de données relationnelles***, dont la gestion peut s'avérer très complexe. Ce sera là l'affaire de logiciels très spécialisés tels que *Oracle, IBM DB, Adabas, PostgreSQL, MySQL*, etc. Python est parfaitement capable de dialoguer avec ces systèmes, mais nous laisserons cela pour un peu plus tard (voir : "Gestion d'une base de données", page 223).

Nos ambitions présentes sont plus modestes. Nos données ne se comptent pas encore par centaines de milliers, aussi nous pouvons nous contenter de mécanismes simples pour les enregistrer dans un fichier de taille moyenne, et les en extraire ensuite.

9.2 Travailler avec des fichiers

L'utilisation d'un fichier ressemble beaucoup à l'utilisation d'un livre. Pour utiliser un livre, vous devez d'abord le trouver (à l'aide de son titre), puis l'ouvrir. Lorsque vous avez fini de l'utiliser, vous le refermez. Tant qu'il est ouvert, vous pouvez y lire des informations diverses, et vous pouvez aussi y écrire des annotations, mais généralement vous ne faites pas les deux à la fois. Dans tous les cas, vous pouvez vous situer à l'intérieur du livre, notamment en vous aidant des numéros de pages. Vous lisez la plupart des livres en suivant l'ordre normal des pages, mais vous pouvez aussi décider de consulter n'importe quel paragraphe dans le désordre.

Tout ce que nous venons de dire des livres s'applique aussi aux fichiers informatiques.
Un fichier se compose de données enregistrées sur votre disque dur, sur une disquette ou sur un CD-ROM. Vous y accédez grâce à son nom (lequel peut inclure aussi un nom de répertoire). Vous pouvez toujours considérer le contenu d'un fichier comme une suite de caractères, ce qui signifie que vous pouvez traiter ce contenu, ou une partie quelconque de celui-ci, à l'aide des fonctions servant à traiter les chaînes de caractères.

9.3 Noms de fichiers - Répertoire courant

Pour simplifier les explications qui vont suivre, nous indiquerons seulement des noms simples pour les fichiers que nous allons manipuler. Si vous procédez ainsi dans vos exercices, les fichiers en question seront créés et/ou recherchés par Python ***dans le répertoire courant***. Celui-ci est habituellement le répertoire où se trouve le script lui-même, sauf si vous lancez ce script depuis la fenêtre d'un *shell IDLE*, auquel cas le répertoire courant est défini au lancement de *IDLE* lui-même (Sous *Window$*, la définition de ce répertoire fait partie des propriétés de l'icône de lancement).

Si vous travaillez avec *IDLE*, vous souhaiterez donc certainement forcer Python à changer son répertoire courant, afin que celui-ci corresponde à vos attentes. Pour ce faire, utilisez les commandes suivantes en début de session. (Nous supposons ici que le répertoire visé est le répertoire **/home/jules/exercices** . Vous pouvez franchement utiliser cette syntaxe (c.à.d. des caractères / et non \ en guise de séparateurs : c'est la convention en vigueur dans le monde Unix). Python effectuera automatiquement les conversions nécessaires, suivant que vous travaillez sous *M@c*, *Linux*, ou *Window$*.[36]

```
>>> from os import chdir
>>> chdir("/home/jules/exercices")
```

La première commande importe la fonction **chdir()** du module **os**. Le module **os** contient toute une série de fonctions permettant de dialoguer avec le système d'exploitation (os = *operating system*), quel que soit celui-ci.

La seconde commande provoque le changement de répertoire ("chdir" ≡ "*change directory"*)

Notes :

- Vous avez également la possibilité d'insérer ces commandes en début de script, ou encore d'indiquer le chemin d'accès complet dans le nom des fichiers que vous manipulez, mais cela risque peut-être d'alourdir l'écriture de vos programmes.
- Choisissez de préférence des noms de fichiers courts. Évitez dans toute la mesure du possible les caractères accentués, les espaces et les signes typographiques spéciaux.

9.4 Les deux formes d'importation

Les lignes d'instructions que nous venons d'utiliser sont l'occasion d'expliquer un mécanisme intéressant. Vous savez qu'en complément des fonctions intégrées dans le module de base, Python met à votre disposition une très grande quantité de fonctions plus spécialisées, qui sont regroupées dans des ***modules***. Ainsi vous connaissez déjà fort bien le module ***math*** et le module ***Tkinter***.

Pour utiliser les fonctions d'un module, il suffit de les importer. Mais cela peut se faire de deux manières différentes, comme nous allons le voir ci-dessous. Chacune des deux méthodes présente des avantages et des inconvénients.

Voici un exemple de la première méthode :

```
>>>>>> import os
>>> rep_cour = os.getcwd()
>>> print rep_cour
C:\Python22\essais
```

36 Dans le cas de Window$, vous pouvez également inclure dans ce chemin la lettre qui désigne le périphérique de stockage où se trouve le fichier. Par exemple : "D:/home/jules/exercices".

La première ligne de cet exemple importe ***l'entièreté*** du module **os**, lequel contient de nombreuses fonctions intéressantes pour l'accès au système d'exploitation. La seconde ligne utilise la fonction **getcwd()** du module **os**[37] Comme vous pouvez le constater, la fonction **getcwd()** renvoie le nom du répertoire courant (*getcwd = get current working directory*).

Par comparaison, voici un exemple similaire utilisant la seconde méthode d'importation :

```
>>> from os import getcwd
>>> rep_cour = getcwd()
>>> print rep_cour
C:\Python22\essais
```

Dans ce nouvel exemple, nous n'avons importé du module **os** que la fonction **getcwd()** seule. Importée de cette manière, la fonction s'intègre à notre propre code comme si nous l'avions écrite nous-mêmes. Dans les lignes où nous l'utilisons, il n'est pas nécessaire de rappeler qu'elle fait partie du module **os**.

Nous pouvons de la même manière importer plusieurs fonctions du même module :

```
>>> from math import sqrt, pi, sin, cos
>>> print pi
3.14159265359
>>> print sqrt(5)                 # racine carrée de 5
2.2360679775
>>> print sin(pi/6)               # sinus d'un angle de 30°
0.5
```

Nous pouvons même importer toutes les fonctions d'un module, comme dans :

```
from Tkinter import *
```

Cette méthode d'importation présente l'avantage d'alléger l'écriture du code. Elle présente l'inconvénient (surtout dans sa dernière forme, celle qui importe toutes les fonctions d'un module) d'encombrer l'espace de noms courant. Il se pourrait alors que certaines fonctions importées aient le même nom que celui d'une variable définie par vous-même, ou encore le même nom qu'une fonction importée depuis un autre module. (Si cela se produit, l'un des deux noms en conflit n'est évidemment plus accessible).

Dans les programmes d'une certaine importance, qui font appel à un grand nombre de modules d'origines diverses, il sera donc toujours préférable de privilégier plutôt la première méthode, c.à.d. celle qui utilise des noms pleinement qualifiés.

On fait généralement exception à cette règle dans le cas particulier du module *Tkinter*, parce que les fonctions qu'il contient sont très sollicitées (dès lors que l'on décide d'utiliser ce module).

37 Le point séparateur exprime donc ici une relation d'appartenance. Il s'agit d'un exemple de la ***qualification des noms*** qui sera de plus en plus largement exploitée dans la suite de ce cours. Relier ainsi des noms à l'aide de points est une manière de désigner sans ambiguïté des éléments faisant partie d'ensembles, lesquels peuvent eux-mêmes faire partie d'ensembles plus vastes, etc. Par exemple, l'étiquette **systeme.machin.truc** désigne l'élément **truc**, qui fait partie de l'ensemble **machin**, lequel fait lui-même partie de l'ensemble **systeme**. Nous verrons de nombreux exemples de cette technique de désignation, notamment lors de notre étude des **classes** d'objets.

9.5 Écriture séquentielle dans un fichier

Sous Python, l'accès aux fichiers est assuré par l'intermédiaire d'un "***objet-fichier***" que l'on crée à l'aide de la fonction interne **open()**. Après avoir appelé cette fonction, vous pouvez lire et écrire dans le fichier en utilisant les méthodes spécifiques de cet objet-fichier.

L'exemple ci-dessous vous montre comment ouvrir un fichier "en écriture", y enregistrer deux chaînes de caractères, puis le refermer. Notez bien que si le fichier n'existe pas encore, il sera créé automatiquement. Par contre, si le nom de fichier utilisé concerne un fichier préexistant qui contient déjà des données, les caractères que nous y enregistrons viendront s'ajouter à la suite de ceux qui s'y trouvent déjà. Vous pouvez faire tout cet exercice directement à la ligne de commande :

```
>>> obFichier = open('Monfichier','a')
>>> obFichier.write('Bonjour, fichier !')
>>> obFichier.write("Quel beau temps, aujourd'hui !")
>>> obFichier.close()
>>>
```

Notes :

- La première ligne crée l'objet-fichier "obFichier", lequel fait référence à un fichier véritable (sur disque ou disquette) dont le nom sera "Monfichier". ***Ne confondez pas le nom de fichier avec le nom de l'objet-fichier*** qui y donne accès. A la suite de cet exercice, vous pouvez vérifier qu'il s'est bien créé sur votre système (dans le répertoire courant) un fichier dont le nom est "Monfichier" (et vous pouvez en visualiser le contenu à l'aide d'un éditeur quelconque).
- La fonction **open()** attend deux arguments, qui doivent être des chaînes de caractères. Le premier argument est le nom du fichier à ouvrir, et le second est le mode d'ouverture. "**a**" indique qu'il faut ouvrir ce fichier en mode "ajout" (*append*), ce qui signifie que les données à enregistrer doivent être ajoutées à la fin du fichier, à la suite de celles qui s'y trouvent éventuellement déjà. Nous aurions pu utiliser aussi le mode "**w**" (pour *write*), mais lorsqu'on utilise ce mode, Python crée toujours un nouveau fichier (vide), et l'écriture des données commence à partir du début de ce nouveau fichier. S'il existe déjà un fichier de même nom, celui-ci est effacé au préalable.
- La méthode **write()** réalise l'écriture proprement dite. Les données à écrire doivent être fournies en argument. Ces données sont enregistrées dans le fichier les unes à la suite des autres (c'est la raison pour laquelle on parle de fichier à accès ***séquentiel***). Chaque nouvel appel de **write()** continue l'écriture à la suite de ce qui est déjà enregistré.
- La méthode **close()** referme le fichier. Celui-ci est désormais disponible pour tout usage.

9.6 Lecture séquentielle d'un fichier

Vous allez maintenant rouvrir le fichier, mais cette fois "en lecture", de manière à pouvoir y relire les informations que vous avez enregistrées dans l'étape précédente :

```
>>> ofi = open('Monfichier', 'r')
>>> t = ofi.read()
>>> print t
Bonjour, fichier !Quel beau temps, aujourd'hui !
>>> ofi.close()
```

Comme on pouvait s'y attendre, la méthode **read()** lit les données présentes dans le fichier et les transfère dans une variable de type "chaîne" (*string*) . Si on utilise cette méthode sans argument, la totalité du fichier est transférée.

Notes :

- Le fichier que nous voulons lire s'appelle "Monfichier". L'instruction d'ouverture de fichier devra donc nécessairement faire référence à ce nom-là. Si le fichier n'existe pas, nous obtenons un message d'erreur. Exemple :

  ```
  >>> ofi = open('Monficier','r')
  IOError: [Errno 2] No such file or directory: 'Monficier'
  ```

  Par contre, nous ne sommes tenus à aucune obligation concernant le nom à choisir pour l'objet-fichier. C'est un nom de variable quelconque. Ainsi donc, dans notre première instruction, nous avons choisi de créer un objet-fichier "**ofi**", faisant référence au fichier réel "Monfichier", lequel est ouvert en lecture (argument "**r**").

- Les deux chaînes de caractères que nous avions entrées dans le fichier sont à présent accolées en une seule. C'est normal, puisque nous n'avons fourni aucun caractère de séparation lorsque nous les avons enregistrées. Nous verrons un peu plus loin comment enregistrer des lignes de texte distinctes.

- La méthode **read()** peut également être utilisée avec un argument. Celui-ci indiquera combien de caractères doivent être lus, à partir de la position déjà atteinte dans le fichier :

  ```
  >>> ofi = open('Monfichier', 'r')
  >>> t = ofi.read(7)
  >>> print t
  Bonjour
  >>> t = ofi.read(15)
  >>> print t
  , fichier !Quel
  ```

  S'il ne reste pas assez de caractères au fichier pour satisfaire la demande, la lecture s'arrête tout simplement à la fin du fichier :

  ```
  >>> t = ofi.read(1000)
  >>> print t
   beau temps, aujourd'hui !
  ```

  Si la fin du fichier est déjà atteinte, **read()** retourne une chaîne vide :

  ```
  >>> t = ofi.read()
  >>> print t


  >>> ofi.close()
  ```

9.7 L'instruction break pour sortir d'une boucle

Il va de soi que les boucles de programmation s'imposent lorsque l'on doit traiter un fichier dont on ne connaît pas nécessairement le contenu à l'avance. L'idée de base consistera à lire ce fichier morceau par morceau, jusqu'à ce que l'on ait atteint la fin du fichier.

La fonction ci-dessous illustre cette idée. Elle copie l'entièreté d'un fichier, quelle que soit sa taille, en transférant des portions de 50 caractères à la fois :

```
def copieFichier(source, destination):
    "copie intégrale d'un fichier"
    fs = open(source, 'r')
    fd = open(destination, 'w')
    while 1:
        txt = fs.read(50)
        if txt =="":
            break
        fd.write(txt)
    fs.close()
    fd.close()
    return
```

Si vous voulez tester cette fonction, vous devez lui fournir deux arguments : le premier est le nom du fichier original, le second est le nom à donner au fichier qui accueillera la copie. Exemple :

```
copieFichier('Monfichier','Tonfichier')
```

Vous aurez remarqué que la boucle **while** utilisée dans cette fonction est construite d'une manière différente de ce que vous avez rencontré précédemment. Vous savez en effet que l'instruction **while** doit toujours être suivie d'une condition à évaluer ; le bloc d'instructions qui suit est alors exécuté en boucle, aussi longtemps que cette condition reste vraie. Or nous avons remplacé ici la condition à évaluer par une simple constante, et vous savez également[38] que l'interpréteur Python considère comme vraie toute valeur numérique différente de zéro.

Une boucle **while** construite comme nous l'avons fait ci-dessus devrait donc boucler indéfiniment, puisque la condition de continuation reste toujours vraie. Nous pouvons cependant interrompre ce bouclage en faisant appel à l'instruction **break**, laquelle permet éventuellement de mettre en place plusieurs mécanismes de sortie différents pour une même boucle :

```
while <condition 1> :
    --- instructions diverses ---
    if <condition 2> :
        break
    --- instructions diverses ---
    if <condition 3>:
        break
    etc.
```

Dans notre fonction **copieFichier()**, il est facile de voir que l'instruction **break** s'exécutera seulement lorsque la fin du fichier aura été atteinte.

38 Voir page 56 : Véracité/fausseté d'une expression

9.8 Fichiers texte

Un fichier texte est un fichier qui contient des caractères imprimables et des espaces organisés en lignes successives, ces lignes étant séparées les unes des autres par un caractère spécial non-imprimable appelé "marqueur de fin de ligne"[39].

Il est très facile de traiter ce genre de fichiers sous Python. Les instructions suivantes créent un fichier texte de quatre lignes :

```
>>> f = open("Fichiertexte", "w")
>>> f.write("Ceci est la ligne un\nVoici la ligne deux\n")
>>> f.write("Voici la ligne trois\nVoici la ligne quatre\n")
>>> f.close()
```

Notez bien le marqueur de fin de ligne "\n" inséré dans les chaînes de caractères, aux endroits où l'on souhaite séparer les lignes de texte dans l'enregistrement. Sans ce marqueur, les caractères seraient enregistrés les uns à la suite des autres, comme dans les exemples précédents.

Lors des opérations de lecture, les lignes d'un fichier texte peuvent être extraites séparément les unes des autres. La méthode **readline()**, par exemple, ne lit qu'une seule ligne à la fois (en incluant le caractère de fin de ligne) :

```
>>> f = open('Fichiertexte','r')
>>> t = f.readline()
>>> print t
Ceci est la ligne un
>>> print f.readline()
Voici la ligne deux
```

La méthode **readlines()** transfère toutes les lignes restantes dans une liste de chaînes :

```
>>> t = f.readlines()
>>> print t
['Voici la ligne trois\012', 'Voici la ligne quatre\012']
>>> f.close()
```

39 Suivant le système d'exploitation utilisé, le codage correspondant au marqueur de fin de ligne peut être différent. Sous Window$, par exemple, il s'agit d'une séquence de deux caractères (Retour chariot et Saut de ligne), alors que dans les systèmes de type Unix (comme Linux) il s'agit d'un seul saut de ligne, M@COS pour sa part utilisant un seul retour chariot. En principe, vous n'avez pas à vous préoccuper de ces différences. Lors des opérations d'écriture, Python utilise la convention en vigueur sur votre système d'exploitation. Pour la lecture, Python interprète correctement chacune des trois conventions (qui sont donc considérées comme équivalentes).

Remarques :

- La liste apparaît ci-dessus en format brut, avec des apostrophes pour délimiter les chaînes, et les caractères spéciaux sous forme de codes numériques. Vous pourrez bien évidemment parcourir cette liste (à l'aide d'une boucle **while**, par exemple) pour en extraire les chaînes individuelles.
- La méthode **readlines()** permet donc de lire l'entièreté d'un fichier en une instruction seulement. Cela n'est possible toutefois que si le fichier à lire n'est pas trop gros (Puisqu'il est copié intégralement dans une variable, c.à.d. dans la mémoire vive de l'ordinateur, il faut que la taille de celle-ci soit suffisante). Si vous devez traiter de gros fichiers, utilisez plutôt la méthode **readline()** dans une boucle, comme le montrera l'exemple de la page suivante.
- Notez bien que **readline()** est une méthode qui renvoie une chaîne de caractères, alors que la méthode **readlines()** renvoie une liste. A la fin du fichier, **readline()** renvoie une chaîne vide, tandis que **readlines()** renvoie une liste vide.

Le script qui suit vous montre comment créer une fonction destinée à effectuer un certain traitement sur un fichier texte. En l'occurrence, il s'agit ici de recopier un fichier texte en omettant toutes les lignes qui commencent par un caractère '#' :

```
def filtre(source,destination):
    "recopier un fichier en éliminant les lignes de remarques"
    fs = open(source, 'r')
    fd = open(destination, 'w')
    while 1:
        txt = fs.readline()
        if txt =='':
            break
        if txt[0] != '#':
            fd.write(txt)
    fs.close()
    fd.close()
    return
```

Pour appeler cette fonction, vous devez utiliser deux arguments : le nom du fichier original, et le nom du fichier destiné à recevoir la copie filtrée. Exemple :

```
filtre('test.txt', 'test_f.txt')
```

9.9 Enregistrement et restitution de variables diverses

L'argument de la méthode **write()** doit être une chaîne de caractères. Avec ce que nous avons appris jusqu'à présent, nous ne pouvons donc enregistrer d'autres types de valeurs qu'en les transformant d'abord en chaînes de caractères.
Nous pouvons réaliser cela à l'aide de la fonction intégrée **str()** :

```
>>> x = 52
>>> f.write(str(x))
```

Nous verrons plus loin qu'il existe d'autres possibilités pour convertir des valeurs numériques en chaînes de caractères (voir à ce sujet : "Formatage des chaînes de caractères", page 126). Mais la question n'est pas vraiment là. Si nous enregistrons les valeurs numériques en les transformant d'abord en chaînes de caractères, nous risquons de ne plus pouvoir les re-transformer correctement en valeurs numériques lorsque nous allons relire le fichier. Exemple :

```
>>> a = 5
>>> b = 2.83
>>> c = 67
>>> f = open('Monfichier', 'w')
>>> f.write(str(a))
>>> f.write(str(b))
>>> f.write(str(c))
>>> f.close()
>>> f = open('Monfichier', 'r')
>>> print f.read()
52.8367
>>> f.close()
```

Nous avons enregistré trois valeurs numériques. Mais comment pouvons-nous les distinguer dans la chaîne de caractères résultante, lorsque nous effectuons la lecture du fichier ? C'est impossible ! Rien ne nous indique d'ailleurs qu'il y a là trois valeurs plutôt qu'une seule, ou 2, ou 4, ...

Il existe plusieurs solutions à ce genre de problèmes. L'une des meilleures consiste à importer un module Python spécialisé : le module pickle[40]. Voici comment il s'utilise :

```
>>> import pickle
>>> f = open('Monfichier', 'w')
>>> pickle.dump(a, f)
>>> pickle.dump(b, f)
>>> pickle.dump(c, f)
>>> f.close()
>>> f = open('Monfichier', 'r')
>>> t = pickle.load(f)
>>> print t, type(t)
5 <type 'int'>
>>> t = pickle.load(f)
>>> print t, type(t)
2.83 <type 'float'>
>>> t = pickle.load(f)
>>> print t, type(t)
67 <type 'int'>
>>> f.close()
```

40 En anglais, le terme *pickle* signifie "conserver". Le module a été nommé ainsi parce qu'il sert effectivement à enregistrer des données en conservant leur type.

Pour cet exemple, on considère que les variables a, b et c contiennent les mêmes valeurs que dans l'exemple précédent. La fonction **dump()** du module **pickle** attend deux arguments : le premier est la variable à enregistrer, le second est l'objet fichier dans lequel on travaille. La fonction **pickle.load()** effectue le travail inverse, c.à.d. la restitution de chaque variable avec son type.

Vous pouvez aisément comprendre ce que font exactement les fonctions du module **pickle** en effectuant une lecture "classique" du fichier résultant, à l'aide de la méthode **read()** par exemple.

9.10 Gestion des exceptions. Les instructions try – except - else

Les ***exceptions*** sont les opérations qu'effectue un interpréteur ou un compilateur lorsqu'une erreur est détectée au cours de l'exécution d'un programme. En règle générale, l'exécution du programme est alors interrompue, et un message d'erreur plus ou moins explicite est affiché.
Exemple :

```
>>> print 55/0
ZeroDivisionError: integer division or modulo
```

(D'autres informations complémentaires sont affichées, qui indiquent notamment à quel endroit du script l'erreur a été détectée, mais nous ne les reproduisons pas ici).

Le message d'erreur proprement dit comporte deux parties séparées par un double point : d'abord le type d'erreur, et ensuite une information spécifique de cette erreur.

Dans de nombreux cas, il est possible de prévoir à l'avance certaines des erreurs qui risquent de se produire à tel ou tel endroit du programme, et d'inclure à cet endroit des instructions particulières, qui seront activées seulement si ces erreurs se produisent. Dans les langages de niveau élevé comme Python, il est également possible d'associer un mécanisme de surveillance à tout ***un ensemble d'instructions***, et donc de simplifier le traitement des erreurs qui peuvent se produire dans n'importe laquelle de ces instructions.

Un mécanisme de ce type s'appelle en général ***mécanisme de traitement des exceptions.*** Celui de Python utilise l'ensemble d'instructions **try - except – else** , qui permettent d'***intercepter*** une erreur et d'exécuter une portion de script spécifique de cette erreur. Il fonctionne comme suit :

Le bloc d'instructions qui suit directement une instruction **try** est exécuté par Python ***sous réserve.*** Si une erreur survient pendant l'exécution de l'une de ces instructions, alors Python annule cette instruction fautive et exécute à sa place le code inclus dans le bloc qui suit l'instruction **except**. Si aucune erreur ne s'est produite dans les instructions qui suivent **try**, alors c'est le bloc qui suit l'instruction **else** qui est exécuté (si cette instruction est présente). Dans tous les cas, l'exécution du programme peut se poursuivre ensuite avec les instructions ultérieures.

Considérons par exemple un script qui demande à l'utilisateur d'entrer un nom de fichier, lequel fichier étant destiné à être ouvert en lecture. Si le fichier n'existe pas, nous ne voulons pas que le programme se "plante". Nous voulons qu'un avertissement soit affiché, et éventuellement que l'utilisateur puisse essayer d'entrer un autre nom.

```
filename = raw_input("Veuillez entrer un nom de fichier : ")
try:
    f = open(filename, "r")
except:
    print "Le fichier", filename, "est introuvable"
```

Si nous estimons que ce genre de test est susceptible de rendre service à plusieurs endroits d'un programme, nous pouvons aussi l'inclure dans une fonction :

```
def existe(fname):
    try:
        f = open(fname,'r')
        f.close()
        return 1
    except:
        return 0

filename = raw_input("Veuillez entrer le nom du fichier : ")
if existe(filename):
    print "Ce fichier existe bel et bien."
else:
    print "Le fichier", filename, "est introuvable."
```

Il est également possible de faire suivre l'instruction **try** de plusieurs blocs **except**, chacun d'entre eux traitant un type d'erreur spécifique, mais nous ne développerons pas ces compléments ici. Veuillez consulter un ouvrage de référence sur Python si nécessaire.

Exercices :

e 92. Écrivez un script qui permette de créer et de relire aisément un fichier texte. Votre programme demandera d'abord à l'utilisateur d'entrer le nom du fichier. Ensuite il lui proposera le choix, soit d'enregistrer de nouvelles lignes de texte, soit d'afficher le contenu du fichier.
L'utilisateur devra pouvoir entrer ses lignes de texte successives en utilisant simplement la touche <Enter> pour les séparer les unes des autres. Pour terminer les entrées, il lui suffira d'entrer une ligne vide (c.à.d. d'utiliser la touche <Enter> seule).
L'affichage du contenu devra montrer les lignes du fichier séparées les unes des autres de la manière la plus naturelle (les codes de fin de ligne ne doivent pas apparaître).

e 93. Considérons que vous avez à votre disposition un fichier texte contenant des phrases de différentes longueurs. Écrivez un script qui recherche et affiche la phrase la plus longue.

e 94. Écrivez un script qui génère automatiquement un fichier texte contenant les tables de multiplication de 2 à 30 (chacune d'entre elles incluant 20 termes seulement).

e 95. Écrivez un script qui recopie un fichier texte en triplant tous les espaces entre les mots.

e 96. Vous avez à votre disposition un fichier texte dont chaque ligne est la représentation d'une valeur numérique de type réel (mais sans exposants). Par exemple :
14.896
7894.6
123.278
etc.
Écrivez un script qui recopie ces valeurs dans un autre fichier en les arrondissant en nombres entiers (l'arrondi doit être correct).

e 97. Écrivez un script qui compare les contenus de deux fichiers et signale la première différence rencontrée.

e 98. A partir de deux fichiers préexistants A et B, construisez un fichier C qui contienne alternativement un élément de A, un élément de B, un élément de A, ... et ainsi de suite jusqu'à atteindre la fin de l'un des deux fichiers originaux. Complétez ensuite C avec les éléments restant sur l'autre.

e 99. Écrivez un script qui permette d'encoder un fichier texte dont les lignes contiendront chacune les noms, prénom, adresse, code postal et n° de téléphone de différentes personnes (considérez par exemple qu'il s'agit des membres d'un club)

e 100. Écrivez un script qui recopie le fichier utilisé dans l'exercice précédent, en y ajoutant la date de naissance et le sexe des personnes (l'ordinateur devra afficher les lignes une par une, et demander à l'utilisateur d'entrer pour chacune les données complémentaires).

e 101. Considérons que vous avez fait les exercices précédents et que vous disposez à présent d'un fichier contenant les coordonnées d'un certain nombre de personnes. Écrivez un script qui permette d'extraire de ce fichier les lignes qui correspondent à un code postal bien déterminé.

e 102. Modifiez le script de l'exercice précédent, de manière à retrouver les lignes correspondant à des prénoms dont la première lettre est située entre F et M (inclus) dans l'alphabet.

e 103. Écrivez des fonctions qui effectuent le même travail que celles du module pickle (voir page 113). Ces fonctions doivent permettre l'enregistrement de variables diverses dans un fichier texte, en les accompagnant systématiquement d'informations concernant leur format exact.

Chapitre 10 : Approfondir les structures de données

10.1 Le point sur les chaînes de caractères

Nous avons déjà rencontré les chaînes de caractères au chapitre **5**. A la différence des données numériques, qui sont des entités singulières, les chaînes de caractères (ou *string*) constituent ***un type de donnée composite***. Nous entendons par là une entité bien définie qui est faite elle-même d'un ensemble d'entités plus petites, en l'occurrence : les caractères.

Suivant les circonstances, nous serons amenés à traiter une telle donnée composite, tantôt comme un seul objet, tantôt comme une suite ordonnée d'éléments. Dans ce dernier cas, nous souhaiterons probablement pouvoir accéder à chacun de ces éléments à titre individuel.

En fait, les chaînes de caractères font partie d'une catégorie d'objets Python que l'on appelle des *séquences*, et dont font partie aussi les *listes* et les *tuples*. On peut effectuer sur les séquences tout un ensemble d'opérations similaires. Vous en connaissez déjà quelques unes, et nous allons en décrire quelques autres dans les paragraphes suivants.

10.1.1 Concaténation, Répétition

Les chaînes peuvent être **concaténées** avec l'opérateur + et **répétées** avec l'opérateur * :

```
>>> n = 'abc' + 'def'            # concaténation
>>> m = 'zut ! ' * 4             # répétition
>>> print n, m
abcdef zut ! zut ! zut ! zut !
```

Remarquez au passage que les opérateurs + et * peuvent aussi être utilisés pour l'addition et la multiplication lorsqu'ils s'appliquent à des arguments numériques. Le fait que les mêmes opérateurs puissent fonctionner différemment en fonction du contexte dans lequel on les utilise est un mécanisme fort intéressant que l'on appelle ***surcharge des opérateurs***. Dans d'autres langages, la surcharge des opérateurs n'est pas toujours possible : on doit alors utiliser des symboles différents pour l'addition et la concaténation, par exemple.

10.1.2 Indiçage, extraction, longueur

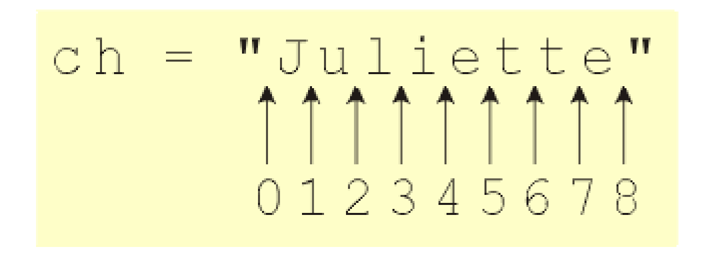
Les chaînes sont des séquences de caractères. Chacun de ceux-ci occupe une place précise dans la séquence. Sous Python, les éléments d'une séquence sont toujours indicés (ou numérotés) de la même manière, c.à.d. ***à partir de zéro***. Pour extraire un caractère d'une chaîne, il suffit d'indiquer son indice ***entre crochets*** :

```
>>> nom = 'Cédric'
>>> print nom[1], nom[3], nom[5]
é r c
```

Il arrive aussi très fréquemment, lorsque l'on travaille avec des chaînes, que l'on souhaite extraire une petite chaîne hors d'une chaîne plus longue. Python propose pour cela une technique simple que l'on appelle ***slicing*** ("découpage en tranches"). Elle consiste à indiquer entre crochets les indices correspondant au début et à la fin de la "tranche" que l'on souhaite extraire :

```
>>> ch = "Juliette"
>>> print ch[0:3]
Jul
```

Dans la tranche **[n,m]**, le n[ième] caractère est inclus, mais pas le m[ième]. Si vous voulez mémoriser aisément ce mécanisme, il faut vous représenter que les indices pointent des emplacements situés entre les caractères, comme dans le schéma ci-dessous :



Au vu de ce schéma, il n'est pas difficile de comprendre que **ch[3:7]** extraira "**iett**"

Les indices de découpage ont des valeurs par défaut : un premier indice non défini est considéré comme zéro, tandis que le second indice omis prend par défaut la taille de la chaîne complète :

```
>>> print ch[:3]       # les 3 premiers caractères
Jul
>>> print ch[3:]       # tout sauf les 3 premiers caractères
iette
```

Exercices

e 104. Déterminez vous-même ce qui se passe lorsque l'un ou l'autre des indices de découpage est erroné, et décrivez cela le mieux possible. (Si le second indice plus petit que le premier, par exemple, ou bien si le second indice est plus grand que la taille de la chaîne).

e 105. Découpez une grande chaîne en fragments de 5 caractères chacun. Rassemblez ces morceaux dans l'ordre inverse.

e 106. Tâchez d'écrire une petite fonction **trouve()** qui fera exactement le contraire de ce que fait l'opérateur d'indexage (c.à.d. les crochets **[]**). Au lieu de partir d'un index donné pour retrouver le caractère correspondant, cette fonction devra retrouver l'index correspondant à un caractère donné.
En d'autres termes, il s'agit d'écrire une fonction qui attend deux arguments : le nom de la chaîne à traiter et le caractère à trouver. La fonction doit fournir en retour l'index du premier caractère de ce type dans la chaîne. Ainsi par exemple, l'instruction :
`print trouve("Juliette & Roméo", "&")` devra afficher : **9**
Attention : Il faut penser à tous les cas possibles. Il faut notamment veiller à ce que la fonction renvoie une valeur particulière (par exemple la valeur ***-1****) si le caractère recherché n'existe pas dans la chaîne traitée.*

e 107. Améliorez la fonction de l'exercice précédent en lui ajoutant un troisième paramètre : l'index à partir duquel la recherche doit s'effectuer dans la chaîne. Ainsi par exemple, l'instruction :
`print trouve ("César & Cléopâtre", "r", 5)` devra afficher : **15** (et non 4 !)

e 108. Écrivez une fonction **comptecar()** qui compte le nombre d'occurrences d'un caractère donné dans une chaîne. Ainsi l'instruction :
`print comptecar("Ananas au jus","a")` devra afficher : **4**

10.1.3 Parcours d'une séquence. L'instruction for ... in ...

Il arrive très souvent que l'on doive traiter l'entièreté d'une chaîne caractère par caractère, du premier jusqu'au dernier, pour effectuer à partir de chacun d'eux une opération quelconque. Nous appellerons cette opération un **parcours**. En nous limitant aux outils Python que nous connaissons déjà, nous pouvons envisager d'encoder un tel parcours sur la base de l'instruction **while** :

```
nom = 'Jacqueline'
index = 0
while index < len(nom):
    print nom[index] + ' *',
    index = index +1
```

Cette boucle "parcourt" donc la chaîne ***nom*** pour en extraire un à un tous les caractères, lesquels sont ensuite imprimés avec interposition d'astérisques. Notez bien que la condition utilisée avec l'instruction while est "**index < len(nom)**", ce qui signifie que le bouclage doit s'effectuer jusqu'à ce que l'on soit arrivé à l'indice numéro 9 (la chaîne compte en effet 10 caractères). Nous aurons bel et bien traité ainsi tous les caractères de la chaîne, puisque ceux-ci sont indicés de ***zéro*** à 9.

Le parcours d'une séquence est une opération très fréquente en programmation. Pour en faciliter l'écriture, Python vous propose une structure de boucle plus appropriée, basée sur le couple d'instructions **for ... in ... :**

Avec ces instructions, le programme ci-dessus devient :

```
nom = 'Jacqueline'
for caract in nom:
    print caract + ' *',
```

Comme vous pouvez le constater, cette structure de boucle est plus compacte. Elle vous évite d'avoir à définir et à incrémenter une variable spécifique (un "compteur") pour gérer l'indice du caractère que vous voulez traiter à chaque itération. La variable **caract** contiendra successivement tous les caractères de la chaîne, du premier jusqu'au dernier.

L'instruction **for** permet donc d'écrire des boucles, dans lesquelles ***l'itération traite successivement tous les éléments d'une séquence donnée***. Dans l'exemple ci-dessus, la séquence était une chaîne de caractères. L'exemple ci-après démontre que l'on peut appliquer le même traitement aux listes (et il en sera de même pour les *tuples* étudiés plus loin) :

```
liste = ['chien','chat','crocodile']
for animal in liste:
    print 'longueur de la chaîne', animal, '=', len(animal)
```

L'exécution de ce script donne :

```
longueur de la chaîne chien = 5
longueur de la chaîne chat = 4
longueur de la chaîne crocodile = 9
```

L'instruction **for** est un nouvel exemple d'***instruction composée***. N'oubliez donc pas le double point obligatoire à la fin de la ligne, et l'indentation du bloc d'instructions qui suit.

Le nom qui suit le mot réservé **in** est celui de la séquence qu'il faut traiter. Le nom qui suit le mot réservé **for** est celui que vous choisissez pour la variable destinée à contenir successivement tous les éléments de la séquence. Cette variable est définie automatiquement (c.à.d. qu'il est inutile de la définir au préalable), et son type est automatiquement adapté à celui de l'élément de la séquence qui est en cours de traitement (rappel : dans le cas d'une liste, tous les éléments ne sont pas nécessairement du même type).

Exemple :

```
divers = ['cheval', 3, 17.25, [5, 'Jean']]
for e in divers:
    print e
```

L'exécution de ce script donne :

```
cheval
3
17.25
[5, 'Jean']
```

Bien que les éléments de la liste **divers** soient tous de types différents (une chaîne de caractères, un entier, un réel, une liste), on peut affecter successivement leurs contenus à la variable **e**, sans qu'il s'ensuive des erreurs (ceci est rendu possible grâce au typage dynamique des variables Python).

Exercices :

e 109. Dans un conte américain, huit petits canetons s'appellent respectivement : Jack, Kack, Lack, Mack, Nack, Oack, Pack et Qack. Écrivez un script qui génère tous ces noms à partir des deux chaînes suivantes :

prefixes = 'JKLMNOP' et **suffixe = 'ack'**

Si vous utilisez une instruction **for ... in ...** , votre script ne devrait comporter que deux lignes.

e 110. Rechercher le nombre de mots contenus dans une phrase donnée.

10.1.4 Appartenance d'un élément à une séquence. L'instruction in utilisée seule

L'instruction **in** peut être utilisée indépendamment de **for**, pour vérifier si un élément donné fait partie ou non d'une séquence. Vous pouvez par exemple vous servir de **in** pour vérifier si tel caractère alphabétique fait partie d'un groupe bien déterminé :

```
car = "e"
voyelles = "aeiouyAEIOUY"
if car in voyelles:
    print car, "est une voyelle"
```

D'une manière similaire, vous pouvez vérifier l'appartenance d'un élément à une liste :

```
n = 5
premiers = [1, 2, 3, 5, 7, 11, 13, 17]
if  n in premiers:
    print n, "fait partie de notre liste de nombres premiers"
```

Exercices :

Note : dans les exercices ci-après, omettez délibérément les caractères accentués et spéciaux.

e 111. Écrivez une fonction **majuscule()** qui retourne "vrai" si l'argument transmis est une majuscule.

e 112. Écrivez une fonction qui retourne "vrai" si l'argument transmis est un chiffre.

e 113. Écrivez une fonction qui convertit une phrase en une liste de mots.

e 114. Utilisez les fonctions définies dans les exercices précédents pour écrire un script qui puisse extraire d'un texte tous les mots qui commencent par une majuscule.

10.1.5 Les chaînes sont des séquences non modifiables

Vous ne pouvez pas modifier le contenu d'une chaîne existante. En d'autres termes, vous ne pouvez pas utiliser l'opérateur **[]** dans la partie gauche d'une instruction d'affectation. Essayez par exemple d'exécuter le petit script suivant (qui cherche à remplacer une lettre dans une chaîne) :

```
salut = 'bonjour à tous'
salut[0] = 'B'
print salut
```

Au lieu d'afficher "Bonjour à tous", ce script "lève" une erreur du genre : *TypeError: object doesn't support item assignment.* Cette erreur est provoquée à la deuxième ligne du script. On y essaie de remplacer une lettre par une autre dans la chaîne, mais cela n'est pas permis.

Par contre, le script ci-dessous fonctionne :

```
salut = 'bonjour à tous'
salut = 'B' + salut[1:]
print salut
```

Dans cet autre exemple, en effet, nous ne modifions pas la chaîne **salut**. Nous en re-créons une nouvelle avec le même nom à la deuxième ligne du script (à partir d'un morceau de la précédente, soit, mais qu'importe : il s'agit bien d'une ***nouvelle*** chaîne).

10.1.6 Les chaînes sont comparables

Tous les opérateurs de comparaison dont nous avons parlé à propos des instructions de contrôle de flux (c.à.d. les instructions **if ... elif ... else**) fonctionnent aussi avec les chaînes de caractères. Cela vous sera très utile pour trier des mots par ordre alphabétique :

```
mot = raw_input("Entrez un mot quelconque : ")
if mot < "limonade":
     place = "précède"
elif mot > "limonade":
     place = "suit"
else:
     place = "se confond avec"
print "Le mot", mot, place, "le mot 'limonade' dans l'ordre alphabétique"
```

Ces comparaisons sont possibles, parce que les caractères alphabétiques qui constituent une chaîne de caractères sont mémorisés dans la mémoire de l'ordinateur sous forme de nombres binaires dont la valeur est liée à la place qu'occupe le caractère dans l'alphabet. Dans le système de codage ASCII, par exemple, A=65, B=66, C=67, etc.[41]

10.1.7 Classement des caractères

Il est souvent utile de pouvoir déterminer si tel caractère extrait d'une chaîne est une lettre majuscule ou minuscule, ou plus généralement encore, de déterminer s'il s'agit bien d'une lettre, d'un chiffre, ou encore d'un autre caractère typographique.

Nous pouvons bien entendu écrire différentes fonctions pour assurer ces tâches. Une première possibilité consiste à utiliser l'instruction **in** comme nous l'avons vu dans un précédent paragraphe. Mais puisque nous savons désormais que les caractères forment une suite bien ordonnée dans le code ASCII, nous pouvons exploiter d'autres méthodes. Par exemple, la fonction ci-dessous retourne "vrai" si l'argument qu'on lui passe est une minuscule :

```
def minuscule(ch):
     if 'a' <= ch <= 'z' :
          return 1
     else:
          return 0
```

41 En fait, il existe plusieurs systèmes de codage : les plus connus sont les codages ASCII et ANSI, assez proches l'un de l'autre sauf en ce qui concerne les caractères particuliers spécifiques des langues autres que l'anglais (caractères accentués, cédilles, etc.). Un nouveau système de codage intégrant tous les caractères spéciaux de toutes les langues mondiales est apparu depuis quelques années. Ce système appelé *unicode* devrait s'imposer petit à petit. Python l'intègre à partir de sa version 2.

Exercices :

Note : dans les exercices ci-après, omettez délibérément les caractères accentués et spéciaux.

e 115. Écrivez une fonction **majuscule()** qui renvoie "vrai" si l'argument transmis est une majuscule (utilisez une autre méthode que celle exploitée précédemment)

e 116. Écrivez une fonction qui renvoie "vrai" si l'argument transmis est un caractère alphabétique quelconque (majuscule ou minuscule). Dans cette nouvelle fonction, utilisez les fonctions **minuscule()** et **majuscule()** définies auparavant.

e 117. Écrivez une fonction qui renvoie "vrai" si l'argument transmis est un chiffre.

e 118. Écrivez une fonction qui renvoie le nombre de caractères majuscules contenus dans une phrase donnée en argument.

Afin que vous puissiez effectuer plus aisément toutes sortes de traitements sur les caractères, Python met à votre disposition un certain nombre de fonctions prédéfinies :

La fonction **ord(ch)** accepte n'importe quel caractère comme argument. En retour, elle fournit le code ASCII correspondant à ce caractère. Ainsi **ord('A')** renvoie la valeur **65**.

La fonction **chr(num)** fait exactement le contraire. L'argument qu'on lui transmet doit être un entier compris entre 0 et 255. En retour, on obtient le caractère ASCII correspondant :
Ainsi **chr(65)** renvoie le caractère **A**.

Exercices :

Note : dans les exercices ci-après, omettez délibérément les caractères accentués et spéciaux.

e 119. Écrivez un petit script qui affiche une table des codes ASCII. Le programme doit afficher tous les caractères en regard des codes correspondants. A partir de cette table, établissez les relations numériques reliant chaque caractère majuscule à chaque caractère minuscule.

e 120. A partir des relations trouvées dans l'exercice précédent, écrivez une fonction qui convertit tous les caractères d'une phrase donnée en minuscules.

e 121. A partir des mêmes relations, écrivez une fonction qui convertit tous les caractères minuscules en majuscules, et vice-versa (dans une phrase fournie en argument).

e 122. Écrivez une fonction qui compte le nombre de fois qu'apparaît tel caractère (fourni en argument) dans une phrase donnée.

e 123. Écrivez une fonction qui renvoie le nombre de voyelles contenues dans une phrase donnée.

10.1.8 Les chaînes sont des objets

Dans les chapitres précédents, vous avez déjà rencontré de nombreux ***objets***. Vous savez donc que l'on peut agir sur un objet à l'aide de ***méthodes*** (c.à.d. des fonctions associées à cet objet).
Sous Python, les chaînes de caractères sont des objets. On peut donc effectuer de nombreux traitements sur les chaînes de caractères en utilisant des méthodes appropriées. En voici quelques-unes, choisies parmi les plus utiles[42] :

- **split() :** convertit une chaîne en une liste de sous-chaînes. On peut choisir le caractère séparateur en le fournissant comme argument, sinon c'est un espace, par défaut :

```
>>> c2 ="Votez pour moi"
>>> a = c2.split()
>>> print a
['Votez', 'pour', 'moi']
>>> c4 ="Cet exemple, parmi d'autres, peut encore servir"
>>> c4.split(",")
['Cet exemple', " parmi d'autres", ' peut encore servir']
```

- **join(liste) :** rassemble une liste de chaînes en une seule (Cette méthode fait donc l'inverse de la précédente). Attention : la chaîne à laquelle on applique cette méthode est celle qui servira de séparateur (un ou plusieurs caractères); l'argument transmis est la liste des chaînes à rassembler :

```
>>> b2 = ["Salut","les","copains"]
>>> print  " ".join(b2)
Salut les copains
>>> print  "---".join(b2)
Salut---les---copains
```

- **find(sch) :** cherche la position d'une sous-chaîne **sch** dans la chaîne :

```
>>> ch1 = "Cette leçon vaut bien un fromage, sans doute ?"
>>> ch2 = "fromage"
>>> print ch1.find(ch2)
25
```

- **count(sch) :** compte le nombre de sous-chaînes **sch** dans la chaîne :

```
>>> ch1 = "Le héron au long bec emmanché d'un long cou"
>>> ch2 = 'long'
>>> print ch1.count(ch2)
2
```

- **lower() :** convertit une chaîne en minuscules :

```
>>> ch ="ATTENTION : Danger !"
>>> print ch.lower()
attention : danger !
```

42 Il s'agit de quelques exemples seulement. La plupart de ces méthodes peuvent être utilisées avec différents paramètres que nous n'indiquons pas tous ici (par exemple, certains paramètres permettent de ne traiter qu'une partie de la chaîne). Vous pouvez obtenir la liste complète de toutes les méthodes associées à un objet à l'aide de la fonction intégrée **dir()**. Veuillez consulter l'un ou l'autre des ouvrages de référence (ou la documentation en ligne) si vous souhaitez en savoir davantage.

- **upper() :** convertit une chaîne en majuscules :

```
>>> ch = "Merci beaucoup"
>>> print ch.upper()
MERCI BEAUCOUP
```

- **capitalize() :** convertit en majuscule la première lettre d'une chaîne :

```
>>> b3 = "quel beau temps, aujourd'hui !"
>>> print b3.capitalize()
"Quel beau temps, aujourd'hui !"
```

- **swapcase() :** convertit toutes les majuscules en minuscules et vice-versa :

```
>>> ch5 = "La CIGALE et la FOURMI"
>>> print ch5.swapcase()
lA cigale ET LA fourmi
```

- **strip() :** enlève les espaces éventuels au début et à la fin de la chaîne :

```
>>> ch = "   Monty Python   "
>>> ch.strip()
'Monty Python'
```

- **replace(c1, c2) :** remplace tous les caractères **c1** par des caractères **c2** dans la chaîne :

```
>>> ch8 = "Si ce n'est toi c'est donc ton frère"
>>> print ch8.replace(" ","*")
Si*ce*n'est*toi*c'est*donc*ton*frère
```

index(c) : retrouve l'index de la première occurrence du caractère **c** dans la chaîne :

```
>>> ch9 ="Portez ce vieux whisky au juge blond qui fume"
>>> print ch9.index("w")
16
```

Dans la plupart de ces méthodes, il est possible de préciser quelle portion de la chaîne doit être traitée, en ajoutant des paramètres supplémentaires. Exemple :

```
>>> print ch9.index("e")          # cherche à partir du début de la chaîne
4                                 # et trouve le premier 'e'
>>> print ch9.index("e",5)        # cherche seulement à partir de l'indice 5
8                                 # et trouve le second 'e'
>>> print ch9.index("e",15)       # cherche à partir du caractère n° 15
29                                # et trouve le quatrième 'e'
```

Etc., etc.

Comprenez bien qu'il n'est pas possible de décrire toutes les méthodes disponibles ainsi que leur paramétrage dans le cadre de ce cours. Si vous souhaitez en savoir davantage, il vous faut consulter la documentation en ligne de Python (Library reference), ou un bon ouvrage de référence (comme par exemple la "Python Standard Library" de Fredrik Lundh – Editions O'Reilly).

Fonctions intégrées

A toutes fins utiles, rappelons également ici que l'on peut aussi appliquer aux chaînes un certain nombre de fonctions intégrées dans le langage lui-même. :

- **len(ch)** renvoie la longueur de la chaîne **ch** (c.à.d. son nombre de caractères)
- **float(ch)** convertit la chaîne **ch** en un nombre réel (*float*)
  (bien entendu, cela ne pourra fonctionner que si la chaîne représente bien un tel nombre) :

```
>>> a = float("12.36")
>>> print a + 5
17.36
```

- **int(ch)** convertit la chaîne **ch** en un nombre entier :

```
>>> a = int("184")
>>> print a + 20
204
```

10.1.9 Formatage des chaînes de caractères

Pour terminer ce tour d'horizon des fonctionnalités associées aux chaînes de caractères, il nous semble utile de vous présenter encore une technique que l'on appelle ***formatage***. Cette technique se révèle particulièrement utile dans tous les cas où vous devez construire une chaîne de caractères complexe à partir d'un certain nombre de morceaux, tels que les valeurs de variables diverses.

Considérons par exemple que vous avez écrit un programme qui traite de la couleur et de la température d'une solution aqueuse, en chimie. La couleur est mémorisée dans une chaîne de caractères nommée **coul**, et la température dans une variable nommée **temp** (variable de type *float)*. Vous souhaitez à présent que votre programme construise une nouvelle chaîne de caractères à partir de ces données, par exemple une phrase telle que la suivante : "La solution est devenue rouge et sa température atteint 12,7 °C".

Vous pouvez construire cette chaîne en assemblant des morceaux à l'aide de l'opérateur de concaténation (le symbole +), mais il vous faudra aussi utiliser la fonction **str()** pour convertir en chaîne de caractères la valeur numérique contenue dans la variable de type *float* (faites l'exercice).

Python vous offre une autre possibilité. Vous pouvez construire votre chaîne en assemblant deux éléments à l'aide de l'opérateur % : à gauche vous fournissez une chaîne de formatage (un patron, en quelque sorte) qui contient des ***marqueurs de conversion***, et à droite (entre parenthèses) un ou plusieurs objets que Python devra insérer dans la chaîne, en lieu et place des marqueurs.

Exemple :

```
>>> coul ="verte"
>>> temp = 1.347 + 15.9
>>> print "La couleur est %s et la température vaut %s °C" % (coul,temp)
La couleur est verte et la température vaut 17.247 °C
```

Dans cet exemple, la chaîne de formatage contient deux marqueurs de conversion `%s` qui seront remplacés respectivement par les contenus des deux variables **coul** et **temp**.

Le marqueur **%s** accepte n'importe quel objet (chaîne, entier, float, ...). Vous pouvez expérimenter d'autres mises en forme en utilisant d'autres marqueurs. Essayez par exemple de remplacer le deuxième **%s** par **%d** , ou par **%8.2f** , ou encore par **%8.2g** . Le marqueur **%d** attend un nombre et le convertit en entier ; les marqueurs **%f** et **%g** attendent des réels et peuvent déterminer la largeur et la précision qui seront affichées.

La description complète de toutes les possibilités de formatage sort du cadre de ces notes. S'il vous faut un formatage très particulier, veuillez consulter la documentation en ligne de Python ou des manuels plus spécialisés.

Exercices :

e 124. Écrivez un script qui compte dans un fichier texte quelconque le nombre de lignes contenant des caractères numériques.

e 125. Écrivez un script qui compte le nombre de mots contenus dans un fichier texte.

e 126. Écrivez un script qui recopie un fichier texte en veillant à ce que chaque ligne commence par une majuscule.

e 127. Écrivez un script qui recopie un fichier texte en fusionnant (avec la précédente) les lignes qui ne commencent pas par une majuscule.

e 128. Vous disposez d'un fichier contenant des valeurs numériques. Considérez que ces valeurs sont les diamètres d'une série de sphères. Écrivez un script qui utilise les données de ce fichier pour en créer un autre, organisé en lignes de texte qui exprimeront "en clair" les autres caractéristiques de ces sphères (surface de section, surface extérieure et volume), dans des phrases telles que :

```
Diam.  46.20 cm Section =  1676.39 cm² Surf. =  6705.54 cm². Vol. =  38724.50 cm³
Diam. 120.00 cm Section = 11309.73 cm² Surf. = 45238.93 cm². Vol. = 678584.01 cm³
Diam.   0.03 cm Section =     0.00 cm² Surf. =     0.00 cm². Vol. =      0.00 cm³
Diam.  13.90 cm Section =   151.67 cm² Surf. =   606.70 cm². Vol. =   1053.89 cm³
Diam.  88.80 cm Section =  6193.21 cm² Surf. = 24772.84 cm². Vol. = 274978.53 cm³
```

etc.

e 129. Vous avez à votre disposition un fichier texte dont les lignes représentent des valeurs numériques de type réel, sans exposant (et encodées sous forme de chaînes de caractères).
Écrivez un script qui recopie ces valeurs dans un autre fichier en les arrondissant de telle sorte que leur partie décimale ne comporte plus qu'un seul chiffre après la virgule, celui-ci ne pouvant être que 0 ou 5 (l'arrondi doit être correct).

10.2 Le point sur les listes

Nous avons déjà rencontré les listes à plusieurs reprises, depuis leur présentation sommaire au chapitre 5. Les listes sont des collections ordonnées d'objets. Comme les chaînes de caractères, les listes font partie d'un type général que l'on appelle ***séquences*** sous Python. Comme les caractères dans une chaîne, les objets placés dans une liste sont rendus accessibles par l'intermédiaire d'un ***index*** (un nombre qui indique l'emplacement de l'objet dans la séquence).

10.2.1 Définition d'une liste – Accès à ses éléments

Vous savez déjà que l'on délimite une liste à l'aide de crochets :

```
>>> nombres = [5, 38, 10, 25]
>>> mots = ["jambon", "fromage", "confiture", "chocolat"]
>>> stuff = [5000, "Brigitte", 3.1416, ["Albert", "René", 1947]]
```

Dans le dernier exemple ci-dessus, nous avons rassemblé un entier, une chaîne, un réel et même une liste, pour vous rappeler que l'on peut combiner dans une liste des données de n'importe quel type, y compris des listes, des dictionnaires et des tuples (ceux-ci seront étudiés plus loin).

Pour accéder aux éléments d'une liste, on utilise les mêmes méthodes (index, découpage en tranches) que pour accéder aux caractères d'une chaîne :

```
>>> print nombres[2]
10
>>> print nombres[1:3]
[38, 10]
>>> print nombres[2:3]
[10]
>>> print nombres[2:]
[10, 25]
>>> print nombres[:2]
[5, 38]
>>> print nombres[-1]
25
>>> print nombres[-2]
10
```

Les exemples ci-dessus devraient attirer votre attention sur le fait qu'une ***tranche*** découpée dans une liste est toujours elle-même une liste (même s'il s'agit d'une tranche qui ne contient qu'un seul élément, comme dans notre troisième exemple), alors qu'un élément isolé peut contenir n'importe quel type de donnée. Nous allons approfondir cette distinction tout au long des exemples suivants.

10.2.2 Les listes sont modifiables

Contrairement aux chaînes de caractères, les listes sont des séquences ***modifiables***. Cela nous permettra de construire plus tard des listes de grande taille, morceau par morceau, d'une manière dynamique (c.à.d. à l'aide d'un algorithme quelconque).

Exemples :

```
>>> nombres[0] = 17
>>> nombres
[17, 38, 10, 25]
```

Dans l'exemple ci-dessus, on a remplacé le premier élément de la liste **nombres**, en utilisant l'opérateur **[ ]** (*opérateur d'indiçage*) à la gauche du signe égale.

Si l'on souhaite accéder à un élément faisant partie d'une liste, elle-même située dans une autre liste, il suffit d'indiquer les deux index entre crochets successifs :

```
>>> stuff[3][1] = "Isabelle"
>>> stuff
[5000, 'Brigitte', 3.1415999999999999, ['Albert', 'Isabelle', 1947]]
```

Comme c'est le cas pour toutes les séquences, il ne faut jamais oublier que la numérotation des éléments commence à partir de zéro. Ainsi, dans l'exemple ci-dessus on remplace l'élément n° 1 d'une liste, qui est elle-même l'élément n° 3 d'une autre liste : la liste **stuff**.

10.2.3 Les listes sont des objets

Sous Python, les listes sont des objets à part entière, et vous pouvez donc leur appliquer un certain nombre de ***méthodes*** particulièrement efficaces :

```
>>> nombres = [17, 38, 10, 25, 72]
>>> nombres.sort()                  # trier la liste
>>> nombres
[10, 17, 25, 38, 72]
>>> nombres.append(12)              # ajouter un élément à la fin
>>> nombres
[10, 17, 25, 38, 72, 12]
>>> nombres.reverse()               # inverser l'ordre des éléments
>>> nombres
[12, 72, 38, 25, 17, 10]
>>> nombres.index(17)               # retrouver l'index d'un élément
4
>>> nombres.remove(38)              # enlever (effacer) un élément
>>> nombres
[12, 72, 25, 17, 10]
```

En plus de ces méthodes, vous disposez encore de l'instruction intégrée **del** , qui vous permet d'effacer un ou plusieurs éléments à partir de leur(s) index :

```
>>> del nombres[2]
>>> nombres
[12, 72, 17, 10]
>>> del nombres[1:3]
>>> nombres
[12, 10]
```

Notez bien la différence entre la méthode **remove()** et l'instruction **del** : **del** travaille avec ***un index*** ou une tranche d'index, tandis que **remove()** recherche ***une valeur*** (si plusieurs éléments de la liste possèdent la même valeur, seul le premier est effacé).

Exercices :

e 130. Écrivez un script qui génère la liste des carrés et des cubes des nombres de 20 à 40.

e 131. Écrivez un script qui crée automatiquement la liste des sinus des angles de 0° à 90° , par pas de 5°. Attention : la fonction **sin()** du module **math** considère que les angles sont fournis en radians (360° = 2 π radians)

e 132. Écrivez un script qui permette d'obtenir à l'écran les 15 premiers termes des tables de multiplication par 2, 3, 5, 7, 11, 13, 17, 19 (ces nombres seront placés au départ dans une liste) sous la forme d'une table similaire à la suivante :

```
2   4   6   8  10  12  14  16  18  20  22  24  26  28  30
3   6   9  12  15  18  21  24  27  30  33  36  39  42  45
5  10  15  20  25  30  35  40  45  50  55  60  65  70  75
etc.
```

e 133. Soit la liste suivante :
['Jean-Michel', 'Marc', 'Vanessa', 'Anne', 'Maximilien', 'Alexandre-Benoît', 'Louise']
Écrivez un script qui affiche chacun de ces noms avec le nombre de caractères correspondant.

e 134. Vous disposez d'une liste de nombres entiers quelconques, certains d'entre eux étant présents en plusieurs exemplaires. Écrivez un script qui recopie cette liste dans une autre, en omettant les doublons. La liste finale devra être triée.

e 135. Écrivez un script qui recherche le mot le plus long dans une phrase donnée (l'utilisateur du programme doit pouvoir entrer une phrase de son choix).

e 136. Écrivez un script capable d'afficher la liste de tous les jours d'une année imaginaire, laquelle commencerait un Jeudi. Votre script utilisera lui-même trois listes : une liste des noms de jours de la semaine, une liste des noms des mois, et une liste des nombres de jours que comportent chacun des mois (ne pas tenir compte des années bissextiles).
Exemple de sortie :

```
Jeudi 1 Janvier   Vendredi 2 Janvier   Samedi 3 Janvier   Dimanche 4 Janvier
```

... et ainsi de suite jusqu'au Jeudi 31 Décembre.

e 137. Vous avez à votre disposition un fichier texte qui contient un certain nombre de noms d'élèves. Écrivez un script qui effectue une copie **triée** de ce fichier.

e 138. Écrivez une fonction permettant de trier une liste. Cette fonction ne pourra pas utiliser la méthode intégrée **sort()** de Python : Vous devez donc définir vous-même l'algorithme de tri.

(Note : cette question devra faire l'objet d'une discussion-synthèse en classe)

10.2.4 Techniques de "slicing" avancé pour modifier une liste

Comme nous venons de le signaler, vous pouvez ajouter ou supprimer des éléments dans une liste en utilisant une instruction (**del**) et une méthode (**append()**) intégrées. Si vous avez bien assimilé le principe du "découpage en tranches" (*slicing*), vous pouvez cependant obtenir les mêmes résultats à l'aide du seul opérateur **[ ]**. L'utilisation de cet opérateur est un peu plus délicate que celle d'instructions ou de méthodes dédiées, mais elle permet davantage de souplesse :

a) Insertion d'un ou plusieurs éléments n'importe où dans une liste

```
>>> mots = ['jambon', 'fromage', 'confiture', 'chocolat']
>>> mots[2:2] =["miel"]
>>> mots
['jambon', 'fromage', 'miel', 'confiture', 'chocolat']
>>> mots[5:5] =['saucisson', 'ketchup']
>>> mots
['jambon', 'fromage', 'miel', 'confiture', 'chocolat', 'saucisson', 'ketchup']
```

Pour utiliser cette technique, vous devez prendre en compte les particularités suivantes :

- Si vous utilisez l'opérateur **[ ]** à la gauche du signe égale pour effectuer une insertion ou une suppression d'élément(s) dans une liste, vous devez obligatoirement y indiquer une "tranche" dans la liste cible (c.à.d. deux index réunis par le symbole :), et non un élément isolé dans cette liste.
- L'élément que vous fournissez à la droite du signe égale doit lui-même être une liste. Si vous n'insérez qu'un seul élément, il vous faut donc le présenter entre crochets pour le transformer d'abord en une liste d'un seul élément. Notez bien que l'élément **mots[1]** n'est pas une liste (c'est la chaîne "fromage"), alors que l'élément **mots[1:3]** en est une.

  Vous comprendrez mieux ces contraintes en analysant ce qui suit :

b) Suppression / remplacement d'éléments

```
>>> mots[2:5] = []                          # [] désigne une liste vide
>>> mots
['jambon','fromage','saucisson', 'ketchup']
>>> mots[1:3] = ['salade']
>>> mots
['jambon', 'salade', 'ketchup']
>>> mots[1:] = ['mayonnaise', 'poulet', 'tomate']
>>> mots
['jambon', 'mayonnaise', 'poulet', 'tomate']
```

- A la première ligne de cet exemple, nous remplaçons la tranche [2:5] par une liste vide, ce qui correspond à un effacement.
- A la quatrième ligne, nous remplaçons une tranche par un seul élément. (Notez encore une fois que cet élément doit lui-même être "présenté" comme une liste).
- A la 7e ligne, nous remplaçons une tranche de deux éléments par une autre qui en comporte 3.

10.2.5 Création d'une liste de nombres à l'aide de la fonction range()

Si vous devez manipuler des séquences de nombres, vous pouvez les créer très aisément à l'aide de cette fonction :

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

La fonction **range()** génère une liste de nombres entiers de valeurs croissantes. Si vous appelez **range()** avec un seul argument, la liste contiendra un nombre de valeurs égal à l'argument fourni, mais en commençant à partir de zéro (*c.à.d. que **range(n)** génère les nombres de **0** à **n-1***).
Notez bien que l'argument fourni n'est jamais dans la liste générée.

On peut aussi utiliser **range()** avec deux, ou même trois arguments séparés par des virgules, afin de générer des séquences de nombres plus spécifiques :

```
>>> range(5,13)
[5, 6, 7, 8, 9, 10, 11, 12]
>>> range(3,16,3)
[3, 6, 9, 12, 15]
```

Si vous avez du mal à assimiler l'exemple ci-dessus, considérez que **range()** attend toujours trois arguments, que l'on pourrait intituler *FROM*, *TO* & *STEP*. *FROM* est la première valeur à générer, *TO* est la dernière (ou plutôt la dernière + un), et *STEP* le "pas" à sauter pour passer d'une valeur à la suivante. S'ils ne sont pas fournis, les paramètres *FROM* et *STEP* prennent leurs valeurs par défaut, qui sont respectivement 0 et 1.

10.2.6 Parcours d'une liste à l'aide de for, range() et len()

L'instruction **for** est l'instruction idéale pour parcourir une liste :

```
>>> prov = ['La','raison','du','plus','fort','est','toujours','la','meilleure']
>>> for mot in prov:
       print mot,
La raison du plus fort est toujours la meilleure
```

Il est très pratique de combiner les fonctions **range()** et **len()** pour obtenir automatiquement tous les indices d'une séquence (liste ou chaîne). Exemple :

```
fable = ['Maître','Corbeau','sur','un','arbre','perché']
for index in range(len(fable)):
    print index, fable[index]
```

L'exécution de ce script donne le résultat :

```
0 Maître
1 Corbeau
2 sur
3 un
4 arbre
5 perché
```

10.2.7 Une conséquence du typage dynamique

Comme nous l'avons déjà signalé plus haut (page 120), le type de la variable utilisée avec l'instruction **for** est redéfini continuellement au fur et à mesure du parcours : même si les éléments d'une liste sont de types différents, on peut parcourir cette liste à l'aide de **for** sans qu'il ne s'ensuive une erreur, car le type de la variable de parcours s'adapte automatiquement à celui de l'élément en cours de lecture. Exemple :

```
>>> divers = [3, 17.25, [5, 'Jean'], 'Linux is not Windoze']
>>> for item in divers:
        print item, type(item)

3 <type 'int'>
17.25 <type 'float'>
[5, 'Jean'] <type 'list'>
Linux is not Windoze <type 'str'>
```

Dans l'exemple ci-dessus, on utilise la fonction intégrée **type()** pour montrer que la variable **item** change effectivement de type à chaque itération (ceci est rendu possible grâce au typage dynamique des variables Python).

10.2.8 Opérations sur les listes

On peut appliquer aux listes les opérateurs + (concaténation) et * (multiplication) :

```
>>> fruits = ['orange','citron']
>>> legumes = ['poireau','oignon','tomate']
>>> fruits + legumes
['orange', 'citron', 'poireau', 'oignon', 'tomate']
>>> fruits * 3
['orange', 'citron', 'orange', 'citron', 'orange', 'citron']
```

L'opérateur * est particulièrement utile pour créer une liste de n éléments identiques :

```
>>> sept_zeros = [0]*7
>>> sept_zeros
[0, 0, 0, 0, 0, 0, 0]
```

Supposons par exemple que vous voulez créer une liste **B** qui contienne le même nombre d'éléments qu'une autre liste **A**. Vous pouvez obtenir ce résultat de différentes manières, mais l'une des plus simples consistera à effectuer : `B = [0]*len(A)`

10.2.9 Test d'appartenance

Vous pouvez aisément déterminer si un élément fait partie d'une liste à l'aide de l'instruction **in** :

```
>>> v = 'tomate'
>>> if v in legumes:
        print 'OK'
OK
```

10.2.10 Copie d'une liste

Considérons que vous souhaitiez réaliser une copie de la liste **fable** définie dans l'exemple précédent, et placer cette copie dans une nouvelle variable que vous appellerez **phrase**. La première idée qui vous viendra à l'esprit sera certainement d'écrire une simple affectation telle que :

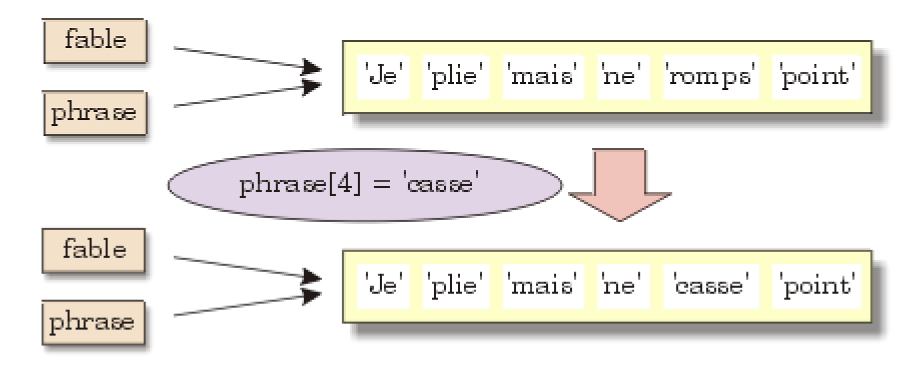
```
>>> phrase = fable
```

En procédant ainsi, sachez que vous ne créez pas une véritable copie. A la suite de cette instruction, il n'existe toujours qu'une seule liste dans la mémoire de l'ordinateur. Ce que vous avez créé est seulement ***une nouvelle référence vers cette liste***. Essayez par exemple :

```
>>> fable = ['Je','plie','mais','ne','romps','point']
>>> phrase = fable
>>> fable[4] ='casse'
>>> phrase
['Je', 'plie', 'mais', 'ne', 'casse', 'point']
```

Si la variable **phrase** contenait une véritable copie de la liste, cette copie serait indépendante de l'original et ne devrait donc pas pouvoir être modifiée par une instruction telle que celle de la troisième ligne, qui s'applique à la variable **fable**. Vous pouvez encore expérimenter d'autres modifications, soit au contenu de **fable**, soit au contenu de **phrase**. Dans tous les cas, vous constaterez que les modifications de l'une sont répercutées dans l'autre, et vice-versa.

En fait, les noms **fable** et **phrase** désignent tous deux un seul et même objet en mémoire. Pour décrire cette situation, les informaticiens diront que le nom **phrase** est un ***alias*** du nom **fable**.



Nous verrons plus tard l'utilité des *alias*. Pour l'instant, nous voudrions disposer d'une technique pour effectuer une véritable copie d'une liste. Avec les notions vues précédemment, vous devriez pouvoir en trouver une par vous-même.

Note :

Python vous autorise à "étendre" une longue instruction sur plusieurs lignes, si vous continuez à encoder quelque chose qui est délimité par une paire de parenthèses, de crochets ou d'accolades. Vous pouvez traiter ainsi des expressions parenthésées, ou encore la définition de longues listes, de grands tuples ou de grands dictionnaires (voir plus loin). Le niveau d'indentation n'a pas d'importance : l'interpréteur détecte la fin de l'instruction, là où la paire syntaxique est refermée.

Cette fonctionnalité vous permet d'améliorer la lisibilité de vos programmes. Exemple :

```
couleurs = ['noir', 'brun', 'rouge',
            'orange', 'jaune', 'vert',
            'bleu', 'violet', 'gris', 'blanc']
```

Exercices :

e 139. Soient les listes suivantes :

```
t1 = [31,28,31,30,31,30,31,31,30,31,30,31]
t2 = ['Janvier','Février','Mars','Avril','Mai','Juin',
 'Juillet','Août','Septembre','Octobre','Novembre','Décembre']
```

Écrivez un petit programme qui insère dans la seconde liste tous les éléments de la première, de telle sorte que chaque nom de mois soit suivi du nombre de jours correspondant : `['Janvier',31,'Février',28,'Mars',31, etc...]`.

e 140. Créez une liste **A** contenant quelques éléments. Effectuez une vraie copie de cette liste dans une nouvelle variable **B**. Suggestion : créez d'abord une liste **B** de même taille que **A** mais ne contenant que des zéros. Remplacez ensuite tous ces zéros par les éléments tirés de **A**.

e 141. Même question, mais autre suggestion : créez d'abord une liste **B** vide. Remplissez-la ensuite à l'aide des éléments de **A** ajoutés l'un après l'autre.

e 142. Même question, autre suggestion encore : pour créer la liste **B**, découpez dans la liste **A** une tranche incluant tous les éléments (à l'aide de l'opérateur **[:]**).

e 143. Un ***nombre premier*** est un nombre qui n'est divisible que par un et par lui-même. Écrivez un programme qui établisse la liste de tous les nombres premiers compris entre 1 et 1000, en utilisant la méthode du ***crible d'Eratosthène*** :
- Créez une liste de 1000 éléments, chacun initialisé à la valeur 1.
- Parcourez cette liste à partir de l'élément d'indice 2 :
  si l'élément analysé possède la valeur 1, mettez à zéro tous les autres éléments de la liste, dont les indices sont des multiples entiers de l'indice auquel vous êtes arrivé.

Lorsque vous aurez parcouru ainsi toute la liste, les indices des éléments qui seront restés à 1 seront les nombres premiers recherchés.
En effet : A partir de l'indice 2, vous annulez tous les éléments d'indices pairs : 4, 6, 8, 10, etc. Avec l'indice 3, vous annulez les éléments d'indices 6, 9, 12, 15, etc., et ainsi de suite. Seuls resteront à 1 les éléments dont les indices sont effectivement des nombres premiers.

10.2.11 Nombres aléatoires - Histogrammes

La plupart des programmes d'ordinateur font exactement la même chose chaque fois qu'on les exécute. De tels programmes sont dits ***déterministes***. Le déterminisme est certainement une bonne chose : nous voulons évidemment qu'une même série de calculs appliquée aux mêmes données initiales aboutisse toujours au même résultat. Pour certaines applications, cependant, nous pouvons souhaiter que l'ordinateur soit imprévisible. Le cas des jeux constitue un exemple évident, mais il en existe bien d'autres.

Contrairement aux apparences, il n'est pas facile du tout d'écrire un algorithme qui soit réellement non-déterministe (c.à.d. qui produise un résultat totalement imprévisible). Il existe cependant des techniques mathématiques permettant de ***simuler*** plus ou moins bien l'effet du hasard. Des livres entiers ont été écrits sur les moyens de produire ainsi un hasard "de bonne qualité". Nous n'allons évidemment pas développer ici une telle question, mais rien ne vous empêche de consulter à ce sujet votre professeur de mathématiques.

Dans son module **random**, Python propose toute une série de fonctions permettant de générer des nombres aléatoires qui suivent différentes distributions mathématiques. Nous n'examinerons ici que quelques-unes d'entre elles. Veuillez consulter la documentation en ligne pour découvrir les autres. Vous pouvez importer toutes les fonctions du module par :

```
>>> from random import *
```

La fonction ci-dessous permet de créer une liste de nombres réels aléatoires, de valeur comprise entre zéro et un. L'argument à fournir est la taille de la liste :

```
>>> def list_aleat(n):
        s = [0]*n
        for i in range(n):
            s[i] = random()
        return s
```

Vous pouvez constater que nous avons pris le parti de construire d'abord une liste de zéros de taille **n**, et ensuite de remplacer les zéros par des nombres aléatoires.

Exercices :

e 144. Réécrivez la fonction **list_aleat()** ci-dessus, en utilisant la méthode **append()** pour construire la liste petit à petit à partir d'une liste vide (au lieu de remplacer les zéros d'une liste préexistante comme nous l'avons fait).

e 145. Ecrivez une fonction **imprime_liste()** qui permette d'afficher ligne par ligne tous les éléments contenus dans une liste de taille quelconque. Le nom de la liste sera fourni en argument. Utilisez cette fonction pour imprimer la liste de nombres aléatoires générés par la fonction **list_aleat()**. Ainsi par exemple, l'instruction **imprime_liste(liste_aleat(8))** devra afficher une colonne de 8 nombres réels aléatoires.

Les nombres ainsi générés sont-ils vraiment aléatoires ? C'est difficile à dire. Si nous ne tirons qu'un petit nombre de valeurs, nous ne pouvons rien vérifier. Par contre, si nous utilisons un grand nombre de fois la fonction **random()**, nous nous attendons à ce que la moitié des valeurs produites soient plus grandes que 0,5 (et l'autre moitié plus petites).

Affinons ce raisonnement. Les valeurs tirées sont toujours dans l'intervalle 0-1. Partageons cet intervalle en 4 fractions égales : de 0 à 0,25 , de 0,25 à 0,5 , de 0,5 à 0,75 , et de 0,75 à 1.
Si nous tirons un grand nombre de valeurs au hasard, nous nous attendons à ce qu'il y en ait autant qui se situent dans chacune de nos 4 fractions. Et nous pouvons généraliser ce raisonnement à un nombre quelconque de fractions, du moment qu'elles soient égales.

Exercice :

e 146. Vous allez écrire un programme destiné à vérifier le fonctionnement du générateur de nombres aléatoires de Python en appliquant la théorie exposée ci-dessus. Votre programme devra donc :

a) Demander à l'utilisateur le nombre de valeurs à tirer au hasard à l'aide de la fonction random(). Il serait intéressant que le programme propose un nombre par défaut (1000 par exemple).

b) Demander à l'utilisateur en combien de fractions il souhaite partager l'intervalle des valeurs possibles (c.à.d. l'intervalle de 0 à 1). Ici aussi, il faudrait proposer un nombre de par défaut (5 fractions, par exemple). Vous pouvez également limiter le choix de l'utilisateur à un nombre compris entre 2 et le 1/10e du nombre de valeurs tirées au hasard.

c) Construire une liste de N compteurs (N étant le nombre de fractions souhaitées). Chacun d'eux sera évidemment initialisé à zéro.

d) Tirer au hasard toutes les valeurs demandées, à l'aide de la fonction **random()** , et mémoriser ces valeurs dans une liste.

e) Mettre en oeuvre un parcours de la liste des valeurs tirées au hasard (boucle), et effectuer un test sur chacune d'elles pour déterminer dans quelle fraction de l'intervalle 0-1 elle se situe. Incrémenter de une unité le compteur correspondant.

f) Lorsque c'est terminé, afficher l'état de chacun des compteurs.

Exemple de résultats affichés par un programme de ce type :

```
Nombre de valeurs à tirer au hasard (défaut = 1000) : 100
Nombre de fractions dans l'intervalle 0-1 (entre 2 et 10, défaut =5) : 5
Tirage au sort des 100 valeurs ...
Comptage des valeurs dans chacune des 5 fractions ...
11 30 25 14 20
```

```
Nombre de valeurs à tirer au hasard (défaut = 1000) : 10000
Nombre de fractions dans l'intervalle 0-1 (entre 2 et 1000, défaut =5) : 5
Tirage au sort des 10000 valeurs ...
Comptage des valeurs dans chacune des 5 fractions ...
1970 1972 2061 1935 2062
```

Une bonne approche de ce genre de problème consiste à essayer d'imaginer quelles fonctions simples vous pourriez écrire pour résoudre l'une ou l'autre partie du problème, puis de les utiliser dans un ensemble plus vaste.

Par exemple, vous pourriez chercher à définir d'abord une fonction **numeroFraction()** qui servirait à déterminer dans quelle fraction de l'intervalle 0-1 une valeur tirée se situe. Cette fonction attendrait deux arguments (la valeur tirée, et le nombre de fractions choisi par l'utilisateur) et fournirait en retour l'index du compteur à incrémenter (c.à.d. le n° de la fraction correspondante). Il existe peut-être un raisonnement mathématique simple qui permette de déterminer l'index de la fraction à partir de ces deux arguments. Pensez notamment à la fonction intégrée **int()** , qui permet de convertir un nombre réel en nombre entier en éliminant sa partie décimale.

Si vous ne trouvez pas, une autre réflexion intéressante serait peut-être de construire d'abord une liste contenant les valeurs "pivots" qui délimitent les fractions retenues (par exemple 0 – 0,25 – 0,5 – 0,75 - 1 dans le cas de 4 fractions). La connaissance de ces valeurs faciliterait peut-être l'écriture de la fonction **numeroFraction()** que nous souhaitons mettre au point.

Si vous disposez d'un temps suffisant, vous pouvez aussi réaliser une version graphique de ce programme, qui présentera les résultats sous la forme d'un histogramme (diagramme "en bâtons").

Tirage au hasard de nombres entiers

Lorsque vous développerez des projets personnels, il vous arrivera fréquemment de souhaiter pouvoir disposer d'une fonction qui permette de tirer au hasard un nombre entier entre certaines limites. Par exemple, si vous voulez écrire un programme de jeu dans lequel des cartes à jouer sont tirées au hasard (à partir d'un jeu ordinaire de 52 cartes), vous aurez certainement l'utilité d'une fonction capable de tirer au hasard un nombre entier compris entre 1 et 52.

Vous pouvez pour ce faire utiliser la fonction **randrange()** du module **random**.

Cette fonction peut être utilisée avec 1, 2 ou 3 arguments.

Avec un seul argument, elle renvoie un entier compris entre zéro et la valeur de l'argument diminué d'une unité. Par exemple, **randrange(6)** retourne un nombre compris entre 0 et 5.

Avec deux arguments, le nombre retourné est compris entre la valeur du premier argument et la valeur du second argument diminué d'une unité. Par exemple, **randrange(2, 8)** retourne un nombre compris entre 2 et 7.

Si l'on ajoute un troisième argument, celui-ci indique que le nombre tiré au hasard doit faire partie d'une série limitée d'entiers, séparés les uns des autres par un certain intervalle défini par ce troisième argument. Par exemple, **randrange(3, 13, 3)** retournera un des nombres de la série 3, 6, 9, 12 :

```
>>> for i in range(15):
        print random.randrange(3,13,3),

3 12 6 9 6 6 12 6 3 6 9 3 6 12 12
```

Exercices :

e 147. Ecrivez un script qui tire au hasard des cartes à jouer. Le nom de la carte tirée doit être correctement présenté, "en clair". Le programme affichera par exemple :

```
Frappez <Enter> pour tirer une carte :
Dix de Trèfle
Frappez <Enter> pour tirer une carte :
As de Carreau
Frappez <Enter> pour tirer une carte :
Huit de Pique
Frappez <Enter> pour tirer une carte :
etc ...
```

10.3 Les tuples

Nous avons étudié jusqu'ici deux types de données composites : les chaînes, qui sont composées de caractères, et les listes, qui sont composées d'éléments de n'importe quel type. Vous devez vous rappeler une autre différence importante entre chaînes et listes : il n'est pas possible de changer les caractères au sein d'une chaîne existante, alors que vous pouvez modifier les éléments d'une liste. En d'autres termes, les listes sont des séquences modifiables, alors que les chaînes sont des séquences non-modifiables. Exemple :

```
>>> liste =['jambon','fromage','miel','confiture','chocolat']
>>> liste[1:3] =['salade']
>>> print liste
['jambon', 'salade', 'confiture', 'chocolat']
>>> chaine ='Roméo préfère Juliette'
>>> chaine[14:] ='Brigitte'
         ***** ==> Erreur: object doesn't support slice assignment  *****
```

Nous essayons de modifier la fin de la chaîne, mais cela ne marche pas. La seule possibilité d'arriver à nos fins est de créer une nouvelle chaîne et d'y recopier ce que nous voulons changer :

```
>>> chaine = chaine[:14] +'Brigitte'
>>> print chaine
Roméo préfère Brigitte
```

Python propose un type de données appelé ***tuple***[43], qui est assez semblable à une liste mais qui n'est pas modifiable.
Du point de vue de la syntaxe, un tuple est une collection d'éléments séparés par des virgules :

```
>>> tuple = 'a', 'b', 'c', 'd', 'e'
>>> print tuple
('a', 'b', 'c', 'd', 'e')
```

Bien que cela ne soit pas nécessaire, il est vivement conseillé de mettre le tuple en évidence en l'enfermant dans une paire de parenthèses, comme l'instruction **print** de Python le fait elle-même.
(Il s'agit simplement d'améliorer la lisibilité du code, mais vous savez que c'est important) :

```
>>> tuple = ('a', 'b', 'c', 'd', 'e')
```

Les opérations que l'on peut effectuer sur des tuples sont syntaxiquement similaires à celles que l'on effectue sur les listes, si ce n'est que les tuples ne sont pas modifiables :

```
>>> print tuple[2:4]
('c', 'd')
>>> tuple[1:3] = ('x', 'y')      ==> ***** erreur *****

>>> tuple = ('André',) + tuple[1:]
>>> print tuple
('André', 'b', 'c', 'd', 'e')
```

Remarquez qu'il faut toujours au moins une virgule pour définir un tuple (le dernier exemple ci-dessus utilise un tuple contenant un seul élément : 'André'). Vous comprendrez l'utilité des tuples petit à petit. Signalons simplement ici qu'ils sont préférables aux listes partout où l'on veut être certain que les données transmises ne soient pas modifiées par erreur au sein d'un programme.

43 ce terme n'est pas un mot anglais : il s'agit d'un néologisme informatique

10.4 Les dictionnaires

Les types composites que nous avons vus jusqu'à présent (chaînes, listes et tuples) utilisent des nombres entiers comme index. A l'aide de ces index, il est facile d'accéder à un élément quelconque de la séquence, mais il faut pour cela connaître à l'avance son emplacement.

Les dictionnaires constituent un autre type composite. Ils ressemblent aux listes dans une certaine mesure (ils sont modifiables comme elles), mais avec cependant une différence très importante au niveau de l'indexage de leurs éléments : dans un dictionnaire, on peut en effet utiliser n'importe quelle structure de données non-modifiable en guise d'index.

10.4.1 Création d'un dictionnaire

A titre d'exemple, nous allons créer un dictionnaire de langue, pour la traduction de termes informatiques anglais en français. Dans ce dictionnaire, les index seront des chaînes de caractères.

Puisque le type dictionnaire est un type modifiable, nous pouvons commencer par créer un dictionnaire vide, puis le remplir petit à petit. Du point de vue syntaxique, on reconnaît une structure de données de type dictionnaire au fait que ses éléments sont enfermés dans **une paire d'accolades**. Un dictionnaire vide sera donc noté **{ }** :

```
>>> dico = {}
>>> dico['computer'] = 'ordinateur'
>>> dico['mouse'] ='souris'
>>> dico['keyboard'] ='clavier'
>>> print dico
{'computer': 'ordinateur', 'keyboard': 'clavier', 'mouse': 'souris'}
```

Comme vous pouvez l'observer dans la ligne ci-dessus, un dictionnaire apparaît comme une série d'éléments séparés par des virgules (le tout étant enfermé entre deux accolades}. Chacun de ces éléments est constitué d'une paire d'objets : un index et une valeur, séparés par un double point.

Dans un dictionnaire, les index s'appellent des ***clés***, et les éléments peuvent donc s'appeler ***des paires clé-valeur***. Vous pouvez constater que l'ordre dans lequel les éléments apparaissent à la dernière ligne ne correspond pas à celui dans lequel nous les avons fournis. Cela n'a strictement aucune importance : nous n'essaierons jamais d'extraire une valeur d'un dictionnaire à l'aide d'un index numérique. Au lieu de cela, nous utiliserons les clés :

```
>>> print dico['mouse']
souris
```

Contrairement à ce qui se passe avec les listes, il n'est pas nécessaire de faire appel à une méthode particulière (telle que append()) pour ajouter de nouveaux éléments à un dictionnaire : il suffit de créer une nouvelle paire clé-valeur.

10.4.2 Opérations sur les dictionnaires

Vous savez déjà comment ajouter des éléments à un dictionnaire. Pour en enlever, il faut utiliser l'instruction **del**. Créons pour l'exemple un autre dictionnaire, destiné cette fois à contenir l'inventaire d'un stock de fruits. Les index (ou clés) seront les noms des fruits, et les valeurs seront les masses de ces fruits répertoriées dans le stock (il s'agit donc cette fois de valeurs de type numérique).

```
>>> invent = {'pommes': 430, 'bananes': 312, 'oranges' : 274, 'poires' : 137}
>>> print invent
{'oranges': 274, 'pommes': 430, 'bananes': 312, 'poires': 137}
```

Si le patron décide de liquider toutes les pommes et de ne plus en vendre, nous pouvons enlever cette entrée dans le dictionnaire :

```
>>> del invent['pommes']
>>> print invent
{'oranges': 274, 'bananes': 312, 'poires': 137}
```

La fonction **len()** est utilisable avec un dictionnaire : elle en retourne le nombre d'éléments.

10.4.3 Les dictionnaires sont des objets

On peut appliquer aux dictionnaires un certain nombre de **méthodes** spécifiques :

La méthode **keys()** retourne la liste des clés utilisées dans le dictionnaire :

```
>>> print dico.keys()
['computer', 'keyboard', 'mouse']
```

La méthode **values()** retourne la liste des valeurs mémorisées dans le dictionnaire :

```
>>> print invent.values()
[274, 312, 137]
```

La méthode **has_key()** permet de savoir si un dictionnaire comprend une clé déterminée. On fournit la clé en argument, et la méthode retourne une valeur 'vraie' ou 'fausse' (en fait, 1 ou 0), suivant que la clé est présente ou pas :

```
>>> print invent.has_key('bananes')
1
>>> if invent.has_key('pommes'):
       print 'nous avons des pommes'
else:
       print 'pas de pommes, sorry'

pas de pommes, sorry
```

La méthode **items()** extrait du dictionnaire une liste équivalente de tuples :

```
>>> print invent.items()
[('oranges', 274), ('bananes', 312), ('poires', 137)]
```

La méthode **copy()** permet d'effectuer une vraie copie d'un dictionnaire. Il faut savoir en effet que la simple affectation d'un dictionnaire existant à une nouvelle variable crée seulement une nouvelle référence vers le même objet, et non un nouvel objet. Nous avons déjà discuté ce phénomène (*aliasing*)à propos des listes (voir page 134). Par exemple, l'instruction ci-dessous ne définit pas un nouveau dictionnaire (contrairement aux apparences) :

```
>>> stock = invent
>>> print stock
{'oranges': 274, 'bananes': 312, 'poires': 137}
```

Si nous modifions **invent**, alors **stock** aussi est modifié, et vice-versa (ces deux noms désignent en effet le même objet dictionnaire dans la mémoire de l'ordinateur) :

```
>>> del invent['bananes']
>>> print stock
{'oranges': 274, 'poires': 137}
```

Pour obtenir une vraie copie (indépendante) d'un dictionnaire préexistant, il faut employer la méthode **copy()** :

```
>>> magasin = stock.copy()
>>> magasin['prunes'] = 561
>>> print magasin
{'oranges': 274, 'prunes': 561, 'poires': 137}
>>> print stock
{'oranges': 274, 'poires': 137}
>>> print invent
{'oranges': 274, 'poires': 137}
```

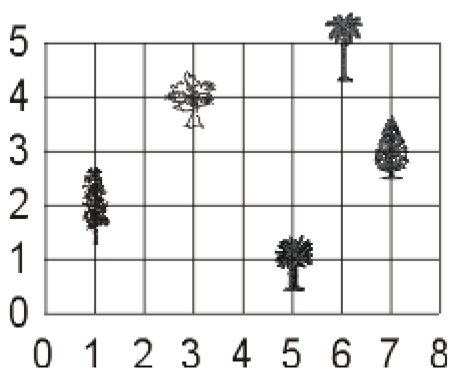
10.4.4 Les clés ne sont pas nécessairement des chaînes de caractères

Jusqu'à présent nous avons décrit des dictionnaires dont les clés étaient à chaque fois des valeurs de type *string*. En fait nous pouvons utiliser en guise de clés n'importe quel type de donnée non modifiable : des entiers, des réels, des chaînes de caractères, et même des tuples.

Considérons par exemple que nous voulions répertorier les arbres remarquables situés dans un grand terrain rectangulaire. Nous pouvons pour cela utiliser un dictionnaire, dont les clés seront des tuples indiquant les coordonnées x,y de chaque arbre :

```
>>> arb = {}
>>> arb[(1,2)] = 'Peuplier'
>>> arb[(3,4)] = 'Platane'
>>> arb[6,5] = 'Palmier'
>>> arb[5,1] = 'Cycas'
>>> arb[7,3] = 'Sapin'

>>> print arb
{(3, 4): 'Platane', (6, 5): 'Palmier', (5, 1):
'Cycas', (1, 2): 'Peuplier', (7, 3): 'Sapin'}
>>> print arb[(6,5)]
palmier
```



Vous pouvez remarquer que nous avons allégé l'écriture à partir de la troisième ligne, en profitant du fait que les parenthèses délimitant les tuples sont facultatives (à utiliser avec prudence !).

Dans ce genre de construction, il faut garder à l'esprit que le dictionnaire contient des éléments seulement pour certains couples de coordonnées. Ailleurs, il n'y a rien. Par conséquent, si nous voulons interroger le dictionnaire pour savoir ce qui se trouve là où il n'y a rien, comme par exemple aux coordonnées (2,1), nous allons provoquer une erreur :

```
>>> print arb[1,2]
Peuplier
>>> print arb[2,1]
                    ***** Erreur : KeyError: (2, 1)  *****
```

Pour résoudre ce petit problème, nous pouvons utiliser la méthode **get()** :

```
>>> print arb.get((1,2),'néant')
Peuplier
>>> print arb.get((2,1),'néant')
néant
```

Le premier argument transmis à cette méthode est la clé de recherche, le second argument est la valeur que nous voulons obtenir en retour si la clé n'existe pas dans le dictionnaire.

10.4.5 Les dictionnaires ne sont pas des séquences

Comme vous l'avez vu plus haut, les éléments d'un dictionnaire ne sont pas disposés dans un ordre particulier. Des opérations comme la concaténation et l'extraction (d'un groupe d'éléments contigus) ne peuvent donc tout simplement pas s'appliquer ici. Si vous essayez tout de même, Python lèvera une erreur lors de l'exécution du code :

```
>>> print arb[1:3]
                    ***** Erreur : KeyError: slice(1, 3, None) *****
```

Vous avez vu également qu'il suffit d'affecter un nouvel indice (une nouvelle clé) pour ajouter une entrée au dictionnaire. Cela ne marcherait pas avec les listes[44] :

```
>>> invent['cerises'] = 987
>>> print invent
{'oranges': 274, 'cerises': 987, 'poires': 137}

>>> liste =['jambon', 'salade', 'confiture', 'chocolat']
>>> liste[4] ='salami'
          ***** IndexError: list assignment index out of range  *****
```

Exercices :

e 148. Créez un dictionnaire qui contienne les noms d'une série d'élèves, leur âge et leur taille. Le nom de l'élève servira de clé d'accès. Exprimez l'âge en années (nombre entier), et exprimez la taille en mètres (ainsi vous devrez employer pour celle-ci une variable de type float !). Écrivez un petit script qui affiche le contenu de ce dictionnaire en utilisant le formatage des chaînes de caractères décrit à la page 126.

e 149. Écrivez une fonction qui échange les clés et les valeurs d'un dictionnaire (ce qui permettra par exemple de transformer un dictionnaire anglais/français en un dictionnaire français/anglais).
(On suppose que le dictionnaire ne contient pas plusieurs valeurs identiques).

44 Rappel : les méthodes permettant d'ajouter des éléments à une liste sont décrites page 131.



10.4.6 Construction d'un histogramme à l'aide d'un dictionnaire

Les dictionnaires constituent un outil très élégant pour construire des histogrammes.

Supposons par exemple que nous voulions établir l'histogramme qui représente la fréquence d'utilisation de chacune des lettres de l'alphabet dans un texte donné. L'algorithme permettant de réaliser ce travail est extraordinairement simple si on le construit sur base d'un dictionnaire :

```
>>> texte ="les saucisses et saucissons secs sont dans le saloir"
>>> lettres ={}
>>> for c in texte:
        lettres[c] = lettres.get(c, 0) + 1

>>> print lettres
{'t': 2, 'u': 2, 'r': 1, 's': 14, 'n': 3, 'o': 3, 'l': 3, 'i': 3, 'd': 1, 'e':
5, 'c': 3, ' ': 8, 'a': 4}
```

Nous commençons par créer un dictionnaire vide : **lettres**. Ensuite, nous allons remplir ce dictionnaire en utilisant les caractères de l'alphabet en guise de clés. Les valeurs que nous mémoriserons pour chacune de ces clés seront les fréquences des caractères correspondants dans le texte. Afin de calculer celles-ci, nous effectuons un parcours de la chaîne de caractères **texte**. Pour chacun de ces caractères, nous interrogeons le dictionnaire à l'aide de la méthode **get()**, en utilisant le caractère en guise de clé, afin d'y lire la fréquence déjà mémorisée pour ce caractère. Si cette valeur n'existe pas encore, la méthode **get()** doit retourner une valeur nulle. Dans tous les cas, nous incrémentons la valeur trouvée, et nous la mémorisons dans le dictionnaire à l'emplacement qui correspond à la clé (c.à.d au caractère en cours de traitement).

Pour fignoler notre travail, nous pouvons encore souhaiter afficher l'histogramme dans l'ordre alphabétique. Pour ce faire, nous pensons immédiatement à la méthode **sort()**, mais celle-ci ne peut s'appliquer qu'aux listes. Qu'à cela ne tienne ! Nous avons vu plus haut comment nous pouvions convertir un dictionnaire en une liste de tuples :

```
>>> lettres_triees = lettres.items()
>>> lettres_triees.sort()
>>> print lettres_triees
[(' ', 8), ('a', 4), ('c', 3), ('d', 1), ('e', 5), ('i', 3), ('l', 3), ('n', 3),
('o', 3), ('r', 1), ('s', 14), ('t', 2), ('u', 2)]
```

Exercices :

e 150. Vous avez à votre disposition un fichier texte quelconque (pas trop gros). Écrivez un script qui compte les occurrences de chacune des lettres de l'alphabet dans ce texte (on ne tiendra pas compte du problème des lettres accentuées)..

e 151. Modifiez le script ci-dessus afin qu'il établisse une table des occurrences de chaque mot dans le texte. Conseil : dans un texte quelconque, les mots ne sont pas seulement séparés par des espaces, mais également par divers signes de ponctuation. Pour simplifier le problème, vous pouvez commencer par remplacer tous les caractères non-alphabétiques par des espaces, et convertir la chaîne résultante en une liste de mots à l'aide de la méthode split().

Chapitre 11 : Classes, objets, attributs

Les chapitres précédents vous ont déjà mis en contact à plusieurs reprises avec la notion d'***objet***. Vous savez donc déjà qu'un objet est une entité que l'on construit par ***instanciation*** à partir d'une ***classe*** (c.à.d. en quelque sorte une "catégorie" ou un "type" d'objet). Par exemple, on peut trouver dans la bibliothèque *Tkinter*, une classe **Button()** à partir de laquelle on peut créer dans une fenêtre un nombre quelconque de boutons.

Nous allons à présent examiner comment vous pouvez vous-mêmes définir de nouvelles classes d'objets. Il s'agit là d'un sujet relativement ardu, mais vous l'aborderez de manière très progressive, en commençant par définir des classes d'objets très simples, que vous perfectionnerez ensuite. Attendez-vous cependant à rencontrer des objets de plus en plus complexes par après.

Comme les objets de la vie courante, les objets informatiques peuvent être très simples ou très compliqués. Ils peuvent être composés de différentes parties, qui soient elles-mêmes des objets, ceux-ci étant faits à leur tour d'autres objets plus simples, etc.

11.1 Utilité des classes

Les classes sont les principaux outils de la programmation orientée objet (*Object Oriented Programming ou OOP*). Ce type de programmation permet de structurer les logiciels complexes en les organisant comme des ensembles d'objets qui interagissent, entre eux et avec le monde extérieur.

Le premier bénéfice de cette approche de la programmation consiste dans le fait que les différents objets utilisés peuvent être construits indépendamment les uns des autres (par exemple par des programmeurs différents) sans qu'il n'y ait de risque d'interférence. Ce résultat est obtenu grâce au concept d'***encapsulation*** : la fonctionnalité interne de l'objet et les variables qu'il utilise pour effectuer son travail, sont en quelque sorte "enfermés" dans l'objet. Les autres objets et le monde extérieur ne peuvent y avoir accès qu'à travers des procédures bien définies.

En particulier, l'utilisation de classes dans vos programmes vous permettra - entre autres choses - ***d'éviter au maximum l'emploi de variables globales***. Vous devez savoir en effet que l'utilisation de variables globales comporte des risques, surtout dans les programmes volumineux, parce qu'il est toujours possible que de telles variables soient modifiées ou même redéfinies n'importe où dans le corps du programme (et ce risque s'aggrave particulièrement si plusieurs programmeurs différents travaillent sur un même logiciel).

Un second bénéfice résultant de l'utilisation des classes est la possibilité qu'elles offrent de ***construire de nouveaux objets à partir d'objets préexistants***, et donc de réutiliser des pans entiers d'une programmation déjà écrite (sans toucher à celle-ci !), pour en tirer une fonctionnalité nouvelle. Cela est rendu possible grâce aux concepts de ***dérivation*** et de ***polymorphisme***.

- La dérivation est le mécanisme qui permet de construire une classe "enfant" au départ d'une classe "parente". L'enfant ainsi obtenu ***hérite*** de toutes les propriétés et de toute la fonctionnalité de son ancêtre, auxquelles on peut ajouter ce que l'on veut.
- Le polymorphisme permet d'attribuer des comportements différents à des objets dérivant les uns des autres, ou au même objet ou en fonction d'un certain contexte.

La programmation orientée objet est optionnelle sous Python. Vous pouvez donc mener à bien de nombreux projets sans l'utiliser, avec des outils plus simples tels que les fonctions. Sachez cependant que les classes constituent des outils pratiques et puissants. Une bonne compréhension des classes vous aidera notamment à maîtriser le domaine des interfaces graphiques (*Tkinter, wxPython*), et vous préparera efficacement à aborder d'autres langages modernes tels que *C++* ou *Java*.

11.2 Définition d'une classe élémentaire

Pour créer une nouvelle classe d'objets Python, on utilise l'instruction **class**.
Nous allons donc apprendre à utiliser cette instruction, en commençant par définir un type d'objet très rudimentaire, lequel sera simplement un nouveau type de donnée. Nous avons déjà utilisé différentes types de données jusqu'à présent, mais c'étaient à chaque fois des types intégrés dans le langage lui-même. Nous allons maintenant créer un nouveau type composite : le type **Point**.

Ce type correspondra au concept de **point** en Mathématique.
Dans un espace à deux dimensions, un point est caractérisé par deux nombres (ses coordonnées suivant x et y). En notation mathématique, on représente donc un point par ses deux coordonnées x et y enfermées dans une paire de parenthèses. On parlera par exemple du point (25,17). Une manière naturelle de représenter un point sous Python serait d'utiliser pour les coordonnées deux valeurs de type *float*. Nous voudrions cependant combiner ces deux valeurs dans une seule entité, ou un seul objet. Pour y arriver, nous allons définir une classe **Point()** :

```
>>> class Point:
        "Définition d'un point mathématique"
```

Les définitions de classes peuvent être situées n'importe où dans un programme, mais on les placera en général au début (ou bien dans un module à importer). L'exemple ci-dessus est probablement le plus simple qui se puisse concevoir. Une seule ligne nous a suffi pour définir le nouveau type d'objet **Point()**. Remarquons d'emblée que :

- L'instruction **class** est un nouvel exemple d'***instruction composée***. N'oubliez pas le double point obligatoire à la fin de la ligne, et l'indentation du bloc d'instructions qui suit. Ce bloc doit contenir au moins une ligne. Dans notre exemple ultra-simplifié, cette ligne n'est rien d'autre qu'un simple commentaire. (Par convention, si la première ligne suivant l'instruction **class** est une chaîne de caractères, celle-ci sera considérée comme un commentaire et incorporée automatiquement dans un dispositif de documentation des classes qui fait partie intégrante de Python. Prenez donc l'habitude de toujours placer une chaîne décrivant la classe à cet endroit).
- Rappelez-vous aussi la convention qui consiste à toujours donner aux classes des noms qui commencent par une majuscule. Dans la suite de ce texte, nous respecterons encore une autre convention qui consiste à associer à chaque nom de classe une paire de parenthèses, comme nous le faisons déjà pour les noms de fonctions.

Nous venons de définir une classe **Point()**. Nous pouvons dès à présent nous en servir pour créer des objets de ce type, par instanciation. Créons par exemple un nouvel objet **p9**[45] :

```
>>> p9 = Point()
```

Après cette instruction, la variable **p9** contient la référence d'un nouvel objet **Point()**.
Nous pouvons dire également que **p9** est ***une nouvelle instance*** de la classe **Point()**.

Attention : comme les fonctions, les classes auxquelles on fait appel dans une instruction doivent toujours être accompagnées de parenthèses (même si aucun argument n'est transmis). Nous verrons un peu plus loin que les classes peuvent être appelées avec des arguments.

Remarquez bien cependant que la ***définition*** d'une classe ne nécessite pas de parenthèses (contrairement à ce qui de règle lors de la définition des fonctions), sauf si nous souhaitons que la classe en cours de définition dérive d'une autre classe préexistante (ceci sera expliqué plus loin).

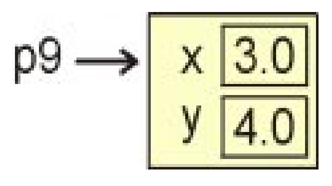
45 Sous Python, on peut donc instancier un objet à l'aide d'une simple instruction d'affectation. D'autres langages imposent l'emploi d'une instruction spéciale, souvent appelée **new** pour bien montrer que l'on crée un nouvel objet à partir d'un moule. Exemple : **p9 = new Point()**

11.3 Attributs (ou variables) d'instance

L'objet que nous venons de créer est une coquille vide. Nous pouvons ajouter des composants à cet objet par simple assignation, en utilisant le système de qualification des noms par points[46] :

```
>>> p9.x = 3.0
>>> p9.y = 4.0
```

Les variables ainsi définies sont des ***attributs*** de l'objet **p9**, ou encore des ***variables d'instance***. Elles sont incorporées, ou plutôt ***encapsulées*** dans l'objet. Le diagramme d'état ci-contre montre le résultat de ces affectations : la variable **p9** contient la référence indiquant l'emplacement mémoire du nouvel objet, qui contient lui-même les deux attributs **x** et **y**.



On peut utiliser les attributs d'un objet dans n'importe quelle expression, comme toutes les variables ordinaires :

```
>>> print p9.x
3.0
>>> print p9.x**2 + p9.y**2
25.0
```

Du fait de leur encapsulation dans l'objet, les attributs sont des variables distinctes d'autres variables qui pourraient porter le même nom. Par exemple, l'instruction **x = p9.x** signifie : "extraire de l'objet référencé par **p9** la valeur de son attribut **x**, et assigner cette valeur à la variable **x**".
Il n'y a pas de conflit entre la variable **x** et l'attribut **x** de l'objet **p9**. L'objet **p9** contient en effet son propre espace de noms, indépendant de l'espace de nom principal où se trouve la variable **x**.

Que se passe-t-il si nous essayons d'afficher l'instance elle-même ?

```
>>> print p9
<__main__.Point instance at 00B1C64C>
```

Ceci indique, comme vous l'aurez certainement bien compris tout de suite, que **p9** est une instance de la classe **Point()** et qu'elle est définie au niveau principal du programme. Elle a reçu de Python un identifiant unique, qui apparaît ici en notation hexadécimale (veuillez consulter votre cours d'informatique générale à ce sujet).

Remarque importante :

Nous venons de voir qu'il est très aisé d'ajouter un attribut à un objet en utilisant une simple instruction d'assignation telle que **p9.x = 3.0** On peut se permettre cela sous Python (c'est une conséquence de l'assignation dynamique des variables), mais ***cela n'est pas vraiment recommandable***, comme vous le comprendrez plus loin. Nous n'utiliserons donc cette façon de faire que de manière occasionnelle, et uniquement dans le but de simplifier nos explications concernant les attributs d'instances.

La bonne manière de procéder sera développée dans le chapitre suivant.

46 Ce système de notation est similaire à celui que nous utilisons pour désigner les variables d'un module, comme par exemple **math.pi** ou **string.uppercase**. Nous aurons l'occasion d'y revenir plus tard, mais sachez dès à présent que les modules peuvent en effet contenir des fonctions, mais aussi des classes et des variables. Essayez par exemple :

```
>>> import  string
>>> print string.uppercase
>>> print string.lowercase
>>> print string.hexdigits
```

11.4 Passage d'objets comme arguments lors de l'appel d'une fonction

Les fonctions peuvent utiliser des objets comme paramètres (elles peuvent également fournir un objet comme valeur de retour). Par exemple, vous pouvez définir une fonction telle que celle-ci :

```
>>> def affiche_point(p):
        print "coord. horizontale =", p.x, "coord. verticale =", p.y
```

Le paramètre **p** utilisé par cette fonction doit être un objet de type **Point()**, puisque l'instruction qui suit utilise les variables d'instance **p.x** et **p.y**. Lorsqu'on appelle cette fonction, il faut donc lui fournir un objet de type **Point()** comme argument. Essayons avec l'objet **p9** :

```
>>> affiche_point(p9)
coord. horizontale = 3.0 coord. verticale = 4.0
```

Exercice :

e 152. Ecrivez une fonction **distance()** qui permette de calculer la distance entre deux points. Cette fonction attendra évidemment deux objets **Point()** comme arguments.

11.5 Similitude et unicité

Dans la langue parlée, les mêmes mots peuvent avoir des significations fort différentes suivant le contexte dans lequel on les utilise. La conséquence en est que certaines expressions utilisant ces mots peuvent être comprises de plusieurs manières différentes (expressions ambiguës).

Le mot "même", par exemple, a des significations différentes dans les phrases : "Charles et moi avons la même voiture" et "Charles et moi avons la même mère". Dans la première, ce que je veux dire est que la voiture de Charles et la mienne sont du même modèle. Il s'agit pourtant de deux voitures distinctes. Dans la seconde, j'indique que la mère de Charles et la mienne constituent en fait une seule et unique personne.

Lorsque nous traitons d'objets logiciels, nous pouvons rencontrer la même ambiguïté. Par exemple, si nous parlons de l'égalité de deux objets **Point()**, cela signifie-t-il que ces deux objets contiennent les mêmes données (leurs attributs), ou bien cela signifie-t-il que nous parlons de deux références à un même et unique objet ?   Considérez par exemple les instructions suivantes :

```
>>> p1 = Point()
>>> p1.x = 3
>>> p1.y = 4
>>> p2 = Point()
>>> p2.x = 3
>>> p2.y = 4
>>> print (p1 == p2)
0
```

Ces instructions créent deux objets **p1** et **p2** qui restent distincts, même s'ils ont des contenus similaires. La dernière instruction teste l'égalité de ces deux objets (double signe égale), et le résultat est zéro (ce qui signifie que l'expression entre parenthèses est fausse : il n'y a donc pas égalité).
On peut confirmer cela d'une autre manière encore :

```
>>> print p1
<__main__.Point instance at 00C2CBEC>
>>> print p2
<__main__.Point instance at 00C50F9C>
```

L'information est claire : les deux variables **p1** et **p2** référencent bien des objets différents.

Essayons autre chose, à présent :

```
>>> p2 = p1
>>> print (p1 == p2)
1
```

Par l'instruction **p2 = p1**, nous assignons le contenu de **p1** à **p2**. Cela signifie que désormais ces deux variables référencent le même objet. Les variables **p1** et **p2** sont des **alias**[47] l'une de l'autre.
Le test d'égalité dans l'instruction suivante retourne cette fois la valeur 1, ce qui signifie que l'expression entre parenthèses est vraie : **p1** et **p2** désignent bien toutes deux un seul et unique objet, comme on peut s'en convaincre en essayant encore :

```
>>> p1.x = 7
>>> print p2.x
7
>>> print p1
<__main__.Point instance at 00C2CBEC>
>>> print p2
<__main__.Point instance at 00C2CBEC>
```

11.6 Objets composés d'objets

Supposons maintenant que nous voulions définir une classe pour représenter des rectangles. Pour simplifier, nous allons considérer que ces rectangles seront toujours orientés horizontalement ou verticalement, et jamais en oblique.

De quelles informations avons-nous besoin pour définir de tels rectangles ?
Il existe plusieurs possibilités. Nous pourrions par exemple spécifier la position du centre du rectangle (deux coordonnées) et préciser sa taille (largeur et hauteur). Nous pourrions aussi spécifier les positions du coin supérieur gauche et du coin inférieur droit. Ou encore la position du coin supérieur gauche et la taille. Admettons ce soit cette dernière méthode qui soit retenue.

Définissons donc notre nouvelle classe :

```
>>> class Rectangle:
        "définition d'une classe de rectangles"
```

... et servons nous-en tout de suite pour créer une instance :

```
>>> boite = Rectangle()
>>> boite.largeur = 50.0
>>> boite.hauteur = 35.0
```

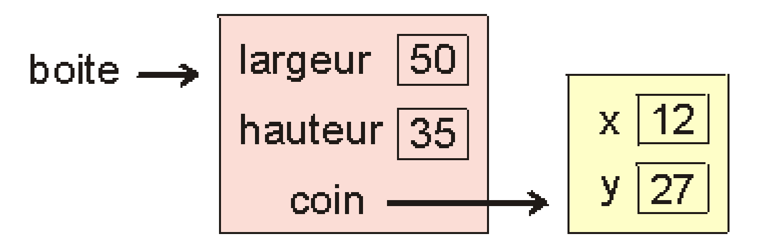
Nous créons ainsi un nouvel objet **Rectangle()** et deux attributs. Pour spécifier le coin supérieur gauche, nous allons utiliser une instance de la classe **Point()** que nous avons définie précédemment. Ainsi nous allons créer un objet à l'intérieur d'un autre objet !

```
>>> boite.coin = Point()
>>> boite.coin.x = 12.0
>>> boite.coin.y = 27.0
```

Pour accéder à un objet qui se trouve à l'intérieur d'un autre objet, on utilise la qualification des noms hiérarchisée (à l'aide de points) que nous avons déjà rencontrée à plusieurs reprises. Ainsi l'expression **boite.coin.y** signifie "Aller à l'objet référencé dans la variable **boite**. Dans cet objet, repérer l'attribut **coin**, puis aller à l'objet référencé dans cet attribut. Une fois cet autre objet trouvé, sélectionner son attribut **y**."

47 Concernant ce phénomène d'aliasing, voir également page 134 : copie d'une liste

Vous pourrez peut-être mieux vous représenter à l'avenir les objets composites, à l'aide de diagrammes similaires à celui que nous reproduisons ci-dessous :



11.7 Objets comme valeurs de retour d'une fonction

Nous avons vu plus haut que les fonctions peuvent utiliser des objets comme paramètres. Elles peuvent également transmettre une instance comme valeur de retour. Par exemple, la fonction **trouveCentre()** ci-dessous doit être appelée avec un argument de type **Rectangle()** et retourne un objet **Point()**, lequel contiendra les coordonnées du centre du rectangle.

```
>>> def trouveCentre(box):
        p = Point()
        p.x = box.coin.x + box.largeur/2.0
        p.y = box.coin.y + box.hauteur/2.0
        return p
```

Pour appeler cette fonction, vous pouvez utiliser l'objet boite comme argument :

```
>>> centre = trouveCentre(boite)
>>> print centre.x, centre.y
37.0  44.5
```

11.8 Les objets sont modifiables

Nous pouvons changer les propriétés d'un objet en assignant de nouvelles valeurs à ses attributs. Par exemple, nous pouvons modifier la taille d'un rectangle (sans modifier sa position), en réassignant ses attributs hauteur et largeur :

```
>>> boite.hauteur = boite.hauteur + 20
>>> boite.largeur = boite.largeur – 5
```

Nous pouvons faire cela sous Python, parce que dans ce langage les propriétés des objets sont toujours publiques (du moins dans la version actuelle 2.0). D'autres langages établissent une distinction nette entre attributs publics (accessibles de l'extérieur de l'objet) et attributs privés (qui sont accessibles seulement aux algorithmes inclus dans l'objet lui-même).

Comme nous l'avons déjà signalé plus haut (à propos de la définition des attributs par assignation simple, depuis l'extérieur de l'objet), ***modifier de cette façon les attributs d'une instance n'est pas une pratique recommandable,*** parce qu'elle contredit l'un des objectifs fondamentaux de la programmation orientée objet, qui vise à établir une séparation stricte entre la fonctionnalité d'un objet (telle qu'elle a été déclarée au monde extérieur) et la manière dont cette fonctionnalité est réellement implémentée dans l'objet (et que le monde extérieur n'a pas à connaître).

Plus concrètement, nous devrons veiller désormais à ce que les objets que nous créons ne soient modifiables en principe que par l'intermédiaire de **méthodes** mises en place spécifiquement dans ce but, comme nous allons l'expliquer dans le chapitre suivant.

Chapitre 12 : Classes, méthodes, héritage

Les classes que nous avons définies dans le chapitre précédent ne sont finalement rien d'autre que des espaces de noms particuliers, dans lesquels nous n'avons placé jusqu'ici que des variables (les attributs d'instance).

Il nous faut à présent doter ces classes d'une fonctionnalité. L'idée de base de la programmation orientée objet consiste en effet à regrouper dans un même ensemble (l'objet) à la fois un certain nombre de données (ce sont les variables ou attributs d'instance) et les algorithmes destinés à effectuer divers traitements sur ces données (ce sont les méthodes, c.à.d. des fonctions encapsulées).

***Objet* = [*attributs* + *méthodes*]**

Cette façon d'associer dans une même "capsule" les propriétés d'un objet et les fonctions qui permettent d'agir sur elles, correspond chez les concepteurs de programmes à une volonté de construire des entités informatiques dont le comportement se rapproche du comportement des objets du monde réel qui nous entoure.

Considérons par exemple un *widget* "bouton". Il nous paraît raisonnable de souhaiter que l'objet informatique que nous appelons ainsi ait un comportement qui ressemble à celui d'un bouton d'appareil quelconque dans le monde réel. Or la fonctionnalité d'un bouton réel (sa capacité de fermer ou d'ouvrir un circuit électrique) est bien intégrée dans l'objet lui-même (au même titre que d'autres propriétés telles que sa taille, sa couleur, etc.) De la même manière, nous souhaiterons que les différentes caractéristiques de notre bouton logiciel (sa taille, son emplacement, sa couleur, le texte qu'il supporte), mais aussi la définition de ce qui se passe lorsque l'on effectue différentes actions de la souris sur ce bouton, soient regroupés dans une entité bien précise à l'intérieur du programme, de manière telle qu'il n'y ait pas de confusion avec un autre bouton ou d'autres entités.

12.1 Définition d'une méthode

Pour illustrer notre propos, nous allons définir une nouvelle classe **Time**, qui nous permettra d'effectuer toute une série d'opérations sur des instants, des durées, etc. :

```
>>> class Time:
        "Définition d'une classe temporelle"
```

Créons à présent un objet de ce type, et ajoutons-lui des variables d'instance pour mémoriser les heures, minutes et secondes :

```
>>> instant = Time()
>>> instant.heure = 11
>>> instant.minute = 34
>>> instant.seconde = 25
```

A titre d'exercice, écrivez maintenant vous-même une fonction **affiche_heure()** , qui serve à visualiser le contenu d'un objet de classe **Time()** sous la forme conventionnelle "heure:minute:seconde".
Appliquée à l'objet **instant** créé ci-dessus, cette fonction devrait donc afficher **11:34:25 :**

```
>>> print affiche_heure(instant)
11:34:25
```

Votre fonction ressemblera probablement à ceci :

```
>>> def affiche_heure(t):
        print str(t.heure) + ":" + str(t.minute) + ":" + str(t.seconde)
```

(Notez au passage l'utilisation de la fonction **str()** pour convertir les données numériques en chaînes de caractères). Si par la suite vous utilisez fréquemment des objets de la classe **Time()**, il y a gros à parier que cette fonction d'affichage vous sera fréquemment utile.

Il serait donc probablement fort judicieux ***d'encapsuler*** cette fonction **affiche_heure()** dans la classe **Time()** elle-même, de manière à s'assurer qu'elle soit toujours automatiquement disponible chaque fois que l'on devra manipuler des objets de la classe **Time()**.

Une fonction qui est ainsi encapsulée dans une classe s'appelle une ***méthode***.

Vous avez déjà rencontré des méthodes à de nombreuses reprises (et vous savez donc déjà qu'une méthode est bien une fonction associée à une classe d'objets).

Définition concrète d'une méthode :

On définit une méthode comme on définit une fonction, avec cependant deux différences :

- ***La définition d'une méthode doit être placée à l'intérieur de la définition d'une classe***, de manière à ce que la relation qui lie la méthode à la classe soit clairement établie.
- ***Le premier paramètre utilisé par une méthode doit toujours être le mot réservé*** " **self** ".
  Ce mot réservé désigne l'instance à laquelle la méthode sera associée, dans les instructions faisant partie de la définition. (La définition d'une méthode comporte donc toujours au moins un paramètre, alors que la définition d'une fonction peut n'en comporter aucun).

Voyons comment cela se passe en pratique :

Pour ré-écrire la fonction **affiche_heure()** comme une méthode de la classe **Time()**, il nous suffit de déplacer sa définition à l'intérieur de celle de la classe, et de changer le nom de son paramètre :

```
>>> class Time:
        "Nouvelle classe temporelle"
        def affiche_heure(self):
            print str(self.heure) + ":" + str(self.minute) \
                  + ":" + str(self.seconde)
```

La définition de la méthode fait maintenant partie du bloc d'instructions indentées après l'instruction **class**. Notez bien l'utilisation du mot réservé **self** , qui se réfère donc à toute instance susceptible d'être créée à partir de cette classe.
(Note : Le code \ permet de continuer une instruction trop longue sur la ligne suivante).

Essai de la méthode dans une instance

Nous pouvons dès à présent instancier un objet de notre nouvelle classe **time()** :

```
>>> maintenant = Time()
```

Si nous essayons d'utiliser un peu trop vite notre nouvelle méthode, ça ne marche pas :

```
>>> maintenant.affiche_heure()
AttributeError: 'Time' instance has no attribute 'heure'
```

C'est normal : nous n'avons pas encore créé les attributs d'instance. Il faudrait faire par exemple :

```
>>> maintenant.heure = 13
>>> maintenant.minute = 34
>>> maintenant.seconde = 21
>>> maintenant.affiche_heure()
13:34:21
```

Nous avons cependant déjà signalé à plusieurs reprises qu'il n'est pas recommandable de créer ainsi les attributs d'instance en dehors de l'objet lui-même, ce qui conduit (entre autres désagréments) à des erreurs comme celle que nous venons de rencontrer, par exemple.

Voyons donc à présent comment nous pouvons mieux faire.

12.2 La méthode "constructeur"

L'erreur que nous avons rencontrée au paragraphe précédent est-elle évitable ?.
Elle ne se produirait effectivement pas, si nous nous étions arrangés pour que la méthode **affiche_heure()** puisse toujours afficher quelque chose, sans qu'il ne soit nécessaire d'effectuer au préalable aucune manipulation sur l'objet nouvellement créé. En d'autres termes, ***il serait judicieux que les variables d'instance soient prédéfinies elles aussi à l'intérieur de la classe***, avec pour chacune d'elles une valeur "par défaut".

Pour obtenir cela, nous allons faire appel à une méthode particulière, que l'on appelle ***un constructeur***. Une méthode constructeur est une méthode qui est exécutée automatiquement lorsque l'on instancie un nouvel objet à partir de la classe. On peut y placer tout ce qui semble nécessaire pour initialiser automatiquement l'objet que l'on crée. Sous Python, la méthode constructeur doit obligatoirement s'appeler **__init__** (deux caractères "souligné", le mot **init**, puis encore deux caractères "souligné").

Exemple :

```
>>> class Time:
        "Encore une nouvelle classe temporelle"
        def __init__(self):
            self.heure =0
            self.minute =0
            self.seconde =0

        def affiche_heure(self):
            print str(self.heure) + ":" + str(self.minute) \
                 + ":" + str(self.seconde)

>>> tstart = Time()
>>> tstart.affiche_heure()
0:0:0
```

L'intérêt de cette technique apparaîtra plus clairement si nous ajoutons encore quelque chose. Comme toute méthode qui se respecte, la méthode **__init__()** peut être dotée de paramètres. Ceux-ci vont jouer un rôle important, parce qu'ils vont permettre d'instancier un objet et d'initialiser certaines de ses variables d'instance, en une seule opération. Dans l'exemple ci-dessus, veuillez donc modifier la définition de la méthode **__init__()** comme suit :

```
        def __init__(self, hh =0, mm =0, ss =0):
            self.heure = hh
            self.minute = mm
            self.seconde = ss
```

La méthode **__init__()** comporte à présent 3 paramètres, avec pour chacun une valeur par défaut. Pour lui transmettre les arguments correspondants, il suffit de placer ceux-ci dans les parenthèses qui accompagnent le nom de la classe, lorsque l'on écrit l'instruction d'instanciation du nouvel objet.

Voici par exemple la création et l'initialisation simultanées d'un nouvel objet **Time()** :

```
>>> recreation = Time(10, 15, 18)
>>> recreation.affiche_heure()
10:15:18
```

Puisque les variables d'instance possèdent maintenant des valeurs par défaut, nous pouvons aussi bien créer de tels objets **Time()** en omettant un ou plusieurs arguments :

```
>>> rentree = Time(10, 30)
>>> rentree.affiche_heure()
10:30:0
```

Exercices :

e 153. Définissez une classe **Domino()** qui permette d'instancier des objets simulant les pièces d'un jeu de dominos. Le constructeur de cette classe initialisera les valeurs des points présents sur les deux faces A et B du domino (valeurs par défaut = 0).
Deux autres méthodes seront définies :
une méthode **affiche_points()** qui affiche les points présents sur les deux faces
une méthode **valeur()** qui renvoie la somme des points présents sur les 2 faces.

Exemples d'utilisation de cette classe :

```
>>> d1 = Domino(2,6)
>>> d2 = Domino(4,3)
>>> d1.affiche_points()
face A : 2  face B : 6
>>> d2.affiche_points()
face A : 4  face B : 3
print "total des points :", d1.valeur() + d2.valeur()
15
>>> liste_dominos = []
>>> for i in range(7):
        liste_dominos.append(Domino(6, i))

>>> print liste_dominos

          etc., etc.
```

e 154. Définissez une classe **CompteBancaire()**, qui permette d'instancier des objets tels que **compte1**, **compte2**, etc. Le constructeur de cette classe initialisera deux attributs d'instance nom et solde, avec les valeurs par défaut '**Dupont**' et **1000**.
Trois autres méthodes seront définies :
- **depot(somme)** permettra d'ajouter une certaine somme au solde
- **retrait(somme)** permettra de retirer une certaine somme du solde
- **affiche()** permettra d'afficher le nom du titulaire et le solde de son compte.

Exemples d'utilisation de cette classe :

```
>>> compte1 = CompteBancaire('Duchmol', 800)
>>> compte1.depot(350)
>>> compte1.retrait(200)
>>> compte1.affiche()
Le solde du compte bancaire de Duchmol est de 950 euros.
>>> compte2 = CompteBancaire()
>>> compte2.depot(25)
>>> compte2.affiche()
Le solde du compte bancaire de Dupont est de 1025 euros.
```

e 155. Définissez une classe **Voiture()** qui permette d'instancier des objets reproduisant le comportement de voitures automobiles. Le constructeur de cette classe initialisera les attributs d'instance suivants, avec les valeurs par défaut indiquées :
marque = '**Ford**', **couleur** = '**rouge**', **pilote** = '**personne**', **vitesse** = **0**.
Lorsque l'on instanciera un nouvel objet **Voiture()**, on pourra choisir sa marque et sa couleur, mais pas sa vitesse, ni le nom de son conducteur.
Les méthodes suivantes seront définies :
- **choix_conducteur(nom)** permettra de désigner (ou changer) le nom du conducteur
- **accelerer(taux, duree)** permettra de faire varier la vitesse de la voiture. La variation de vitesse obtenue sera égale au produit : **taux** x **duree**. Par exemple, si la voiture accélère au taux de 1,3 m/s^2 pendant 20 secondes, son gain de vitesse doit être égal à 26 m/s. Des taux négatifs seront acceptés (ce qui permettra de décélérer). La variation de vitesse ne sera pas autorisée si le conducteur est 'personne'.
- **affiche_tout()** permettra de faire apparaître les propriétés présentes de la voiture, c.à.d. sa marque, sa couleur, le nom de son conducteur, sa vitesse.

Exemples d'utilisation de cette classe :

```
>>> a1 = Voiture('Peugeot', 'bleue')
>>> a2 = Voiture(couleur = 'verte')
>>> a3 = Voiture('Mercedes')
>>> a1.choix_conducteur('Roméo')
>>> a2.choix_conducteur('Juliette')
>>> a2.accelerer(1.8, 12)
>>> a3.accelerer(1.9, 11)
Cette voiture n'a pas de conducteur !
>>> a2.affiche_tout()
Ford verte pilotée par Juliette, vitesse = 21.6 m/s.
>>> a3.affiche_tout()
Mercedes rouge pilotée par personne, vitesse = 0 m/s.
```

e 156. Définissez une classe **Satellite()** qui permette d'instancier des objets simulant des satellites artificiels lancés dans l'espace, autour de la terre. Le constructeur de cette classe initialisera les attributs d'instance suivants, avec les valeurs par défaut indiquées :

masse = **100**, **vitesse** = **0**.

Lorsque l'on instanciera un nouvel objet **Satellite()**, on pourra choisir son nom, sa masse et sa vitesse.

Les méthodes suivantes seront définies :

- **impulsion(force, duree)** permettra de faire varier la vitesse du satellite. Pour savoir comment, rappelez-vous votre cours de physique : la variation de vitesse Δv subie par un objet de masse ***m*** soumis à l'action d'une force ***F*** pendant un temps ***t*** vaut $\Delta v = \frac{F \times t}{m}$. Par exemple : un satellite de 300 kg qui subit une force de 600 Newtons pendant 10 secondes voit sa vitesse augmenter (ou diminuer) de 20 m/s.
- **affiche_vitesse()** affichera le nom du satellite et sa vitesse courante.
- **energie()** renverra au programme appelant la valeur de l'énergie cinétique du satellite.

Rappel : l'énergie cinétique se calcule à l'aide de la formule $E_c = \frac{m \times v^2}{2}$

Exemples d'utilisation de cette classe :

```
>>> s1 = Satellite('Zoé', masse =250, vitesse =10)
>>> s1.impulsion(500, 15)
>>> s1.affiche_vitesse()
vitesse du satellite Zoé = 40 m/s.
>>> print s1.energie()
200000
>>> s1.impulsion(500, 15)
>>> s1.affiche_vitesse()
vitesse du satellite Zoé = 70 m/s.
>>> print s1.energie()
612500
```

12.3 Espaces de noms des classes et instances

Vous avez appris précédemment (voir page 67) que les variables définies à l'intérieur d'une fonction sont des variables locales, inaccessibles aux instructions qui se trouvent à l'extérieur de la fonction. Cela vous permet d'utiliser les mêmes noms de variables dans différentes parties d'un programme, sans risque d'interférence.

Pour décrire la même chose en d'autres termes, nous pouvons dire que chaque fonction possède son propre ***espace de noms***, indépendant de l'espace de noms principal.

Vous avez appris également que les instructions se trouvant à l'intérieur d'une fonction peuvent accéder aux variables définies au niveau principal, mais ***en lecture seulement*** : elles peuvent utiliser les valeurs de ces variables, mais pas les modifier (à moins de faire appel à l'instruction **global**).

Il existe donc une sorte de hiérarchie entre les espaces de noms. Nous allons constater la même chose à propos des classes et des objets. En effet :

- Chaque classe possède son propre espace de noms. Les variables qui en font partie sont appelées les attributs de la classe.
- Chaque objet instance (créé à partir d'une classe) obtient son propre espace de noms. Les variables qui en font partie sont appelées variables d'instance ou attributs d'instance.
- Les classes peuvent utiliser (mais pas modifier) les variables définies au niveau principal.
- Les instances peuvent utiliser (mais pas modifier) les variables définies au niveau de la classe et les variables définies au niveau principal.

Considérons par exemple la classe **Time()** définie précédemment. A la page 154, nous avons instancié deux objets de cette classe : **recreation** et **rentree**. Chacun a été initialisé avec des valeurs différentes, indépendantes. Nous pouvons modifier et réafficher ces valeurs à volonté dans chacun de ces deux objets, sans que l'autre n'en soit affecté :

```
>>> recreation.heure = 12
>>> rentree.affiche_heure()
10:30:0
>>> recreation.affiche_heure()
12:15:18
```

Veuillez à présent encoder et tester l'exemple ci-dessous :

```
>>> class Espaces:                          # 1
        aa = 33                             # 2
        def affiche(self):                  # 3
            print aa, Espaces.aa, self.aa   # 4

>>> aa = 12                                 # 5
>>> essai = Espaces()                       # 6
>>> essai.aa = 67                           # 7
>>> essai.affiche()                         # 8
12 33 67
>>> print a, Espaces.a, essai.a             # 9
12 33 67
```

Dans cet exemple, le même nom **aa** est utilisé pour définir trois variables différentes : une dans l'espace de noms de la classe (à la ligne 2), une autre dans l'espace de noms principal (à la ligne 5), et enfin une dernière dans l'espace de nom de l'instance (à la ligne 7).
La ligne 4 et la ligne 9 montrent comment vous pouvez accéder à ces trois espaces de noms (de l'intérieur d'une classe, ou au niveau principal), en utilisant la qualification par points. Notez encore une fois l'utilisation de **self** pour désigner l'instance.

12.4 Héritage

Les classes constituent le principal outil de la programmation orientée objet (*Object Oriented Programming ou OOP*), qui est considérée de nos jours comme la technique de programmation la plus performante. L'un des principaux atouts de ce type de programmation réside dans le fait que l'on peut toujours se servir d'une classe préexistante pour en créer une nouvelle qui possédera quelques fonctionnalités supplémentaires. Le procédé s'appelle ***dérivation***. Il permet de créer toute une hiérarchie de classes allant du général au particulier.

Nous pouvons par exemple définir une classe **Mammifere()**, qui contiendra un ensemble de caractéristiques propres à ce type d'animal. A partir de cette classe, nous pourrons alors dériver une classe **Primate()**, une classe **Rongeur()**, une classe **Carnivore()**, etc., qui ***hériteront*** de toutes les caractéristiques de la classe **Mammifere()**, en y ajoutant leurs spécificités.

Au départ de la classe **Carnivore()**, nous pourrons ensuite dériver une classe **Belette()**, une classe **Loup()**, une classe **Chien()**, etc., qui hériteront encore une fois de toutes les caractéristiques de la classe parente avant d'y ajouter les leurs. **Exemple :**

```
>>> class Mammifere:
        caract1 = "il allaite ses petits ;"

>>> class Carnivore(Mammifere):
        caract2 = "il se nourrit de la chair de ses proies ;"

>>> class Chien(Carnivore):
        caract3 = "son cri s'appelle aboiement ;"

>>> mirza = Chien()
>>> print mirza.caract1, mirza.caract2, mirza.caract3
il allaite ses petits ; il se nourrit de la chair de ses proies ;
son cri s'appelle aboiement ;
```

Dans cet exemple, nous voyons que l'objet **mirza** , qui est une instance de la classe **Chien()**, hérite non seulement de l'attribut défini pour cette classe, mais également des attributs définis pour les classes parentes.

Vous voyez également dans cet exemple comment il faut procéder pour dériver une classe à partir d'une classe parente : On utilise l'instruction **class** , suivie comme d'habitude du nom que l'on veut attribuer à la nouvelle classe, ***et on place entre parenthèses le nom de la classe parente***.

Notez bien que les attributs utilisés dans cet exemple sont des attributs des classes (et non des attributs d'instances). L'instance **mirza** peut accéder à ces attributs, mais pas les modifier :

```
>>> mirza.caract2 = "son corps est couvert de poils"     # 1
>>> print mirza.caract2                                  # 2
son corps est couvert de poils                           # 3
>>> fido = Chien()                                       # 4
>>> print fido.caract2                                   # 5
il se nourrit de la chair de ses proies ;                # 6
```

Dans ce nouvel exemple, la ligne 1 ne modifie pas l'attribut **caract2** de la classe **Carnivore()**, contrairement à ce que l'on pourrait penser au vu de la ligne 3. Nous pouvons le vérifier en créant une nouvelle instance **fido** (lignes 4 à 6) .

Si vous avez bien assimilé les paragraphes précédents, vous aurez compris que l'instruction de la ligne 1 crée une nouvelle variable d'instance associée seulement à l'objet **mirza**. Il existe donc dès ce moment deux variables avec le même nom **caract2** : l'une dans l'espace de noms de l'objet **mirza**, et l'autre dans l'espace de noms de la classe **Carnivore()**.

Comment faut-il alors interpréter ce qui s'est passé aux lignes 2 et 3 ? Comme nous l'avons vu plus haut, l'instance **mirza** peut accéder aux variables situées dans son propre espace de noms, mais aussi à celles qui sont situées dans les espaces de noms de toutes les classes parentes. S'il existe des variables aux noms identiques dans plusieurs de ces espaces, laquelle sera-t-elle sélectionnée lors de l'exécution d'une instruction comme celle de la ligne 2 ?

Pour résoudre ce conflit, Python respecte une règle de priorité fort simple. Lorsqu'on lui demande d'utiliser la valeur d'une variable nommée *alpha*, par exemple, il commence par rechercher ce nom dans l'espace local (le plus "interne", en quelque sorte). Si une variable *alpha* est trouvée dans l'espace local, c'est celle-là qui est utilisée, et la recherche s'arrête. Sinon, Python examine l'espace de noms de la structure parente, puis celui de la structure grand-parente, et ainsi de suite jusqu'au niveau principal du programme.

A la ligne 2 de notre exemple, c'est donc la variable d'instance qui sera utilisée. A la ligne 5, par contre, c'est seulement au niveau de la classe grand-parente qu'une variable répondant au nom **caract2** peut être trouvée. C'est donc celle-là qui est affichée.

12.5 Héritage et polymorphisme

Analysez soigneusement le script de la page suivante. Il met en oeuvre plusieurs concepts décrits précédemment, en particulier le concept d'héritage.

Pour bien comprendre ce script, il faut cependant d'abord vous rappeler quelques notions élémentaires de ***chimie***. Dans votre cours de chimie, vous avez certainement dû apprendre que les ***atomes*** sont des entités constitués d'un certain nombre de ***protons*** (particules chargées d'électricité positive), d'***électrons*** (chargés négativement) et de ***neutrons*** (neutres).

Le type d'atome (ou élément) est déterminé par le nombre de protons, que l'on appelle également ***numéro atomique***. Dans son état fondamental, un atome contient autant d'électrons que de protons, et par conséquent il est électriquement neutre. Il possède également un nombre variable de neutrons, mais ceux-ci n'influencent en aucune manière la charge électrique globale.

Dans certaines circonstances, un atome peut gagner ou perdre des électrons. Il acquiert de ce fait une charge électrique globale, et devient alors un ***ion*** (il s'agit d'un ***ion négatif*** si l'atome a gagné un ou plusieurs électrons, et d'un ***ion positif*** s'il en a perdu). La charge électrique d'un ion est égale à la différence entre le nombre de protons et le nombre d'électrons qu'il contient.

Le script reproduit à la page suivante génère des objets "atome" et des objets "ion". Nous avons rappelé ci-dessus qu'un ion est simplement un atome modifié. Dans notre programmation, la classe qui définit les objets "ion" sera donc une classe dérivée de la classe qui définit les objets "atome" : elle héritera d'elle tous ses attributs et toutes ses méthodes, en y ajoutant les siennes propres.

L'une de ces méthodes ajoutées (la méthode affiche()) remplace une méthode de même nom héritée de la classe "atome". Les classes "atome" et "ion" possèdent donc chacune une méthode de même nom, mais qui effectuent un travail différent. On parle dans ce cas de ***polymorphisme***.

Il sera évidemment possible d'instancier un nombre quelconque d'atomes et d'ions à partir de ces deux classes. Or l'une d'entre elles (la classe "atome") doit contenir une version simplifiée du tableau périodique des éléments (tableau de Mendeléev), de façon à pouvoir attribuer un nom d'élément chimique, ainsi qu'un nombre de neutrons, à chaque objet généré. Comme il n'est pas souhaitable de recopier tout ce tableau dans chacune des instances, nous le placerons dans ***un attribut de classe***. Ainsi ce tableau n'existera qu'en un seul endroit en mémoire, tout en restant accessible à tous les objets qui seront produits à partir de cette classe.

Voyons concrètement comment toutes ces idées s'articulent :

```
class Atome:
    """atomes simplifiés, choisis parmi les 10 premiers éléments du TP"""
    table =[None, ('hydrogène',0), ('hélium',2), ('lithium',4),
            ('béryllium',5), ('bore',6), ('carbone',6), ('azote',7),
            ('oxygène',8), ('fluor',10), ('néon',10)]

    def __init__(self, nat):
        "le n° atomique détermine le n. de protons, d'électrons et de neutrons"
        self.np, self.ne = nat, nat          # nat = numéro atomique
        self.nn = Atome.table[nat][1]

    def affiche(self):
        print
        print "Nom de l'élément :", Atome.table[self.np][0]
        print "%s protons, %s électrons, %s neutrons" % \
                  (self.np, self.ne, self.nn)

class Ion(Atome):
    """les ions sont des atomes qui ont gagné ou perdu des électrons"""

    def __init__(self, nat, charge):
        "le n° atomique et la charge électrique déterminent l'ion"
        Atome.__init__(self, nat)
        self.ne = self.ne - charge
        self.charge = charge

    def affiche(self):
        "cette méthode remplace celle héritée de la classe parente"
        Atome.affiche(self)          # ... tout en l'utilisant elle-même !
        print "Particule électrisée. Charge =", self.charge

### Programme principal : ###

a1 = Atome(5)
a2 = Ion(3, 1)
a3 = Ion(8, -2)
a1.affiche()
a2.affiche()
a3.affiche()
```

L'exécution de ce script provoque l'affichage suivant :

```
Nom de l'élément : bore
5 protons, 5 électrons, 6 neutrons

Nom de l'élément : lithium
3 protons, 2 électrons, 4 neutrons
Particule électrisée. Charge = 1

Nom de l'élément : oxygène
8 protons, 10 électrons, 8 neutrons
Particule électrisée. Charge = -2
```

Au niveau du programme principal, vous pouvez constater que l'on instancie les objets **Atome()** en fournissant leur numéro atomique (lequel doit être compris entre 1 et 10). Pour instancier des objets **Ion()**, par contre, on doit fournir un numéro atomique et une charge électrique globale (positive ou négative). La même méthode **affiche()** fait apparaître les propriétés de ces objets, qu'il s'agisse d'atomes ou d'ions, avec dans le cas de l'ion une ligne supplémentaire (polymorphisme).

Commentaires :

La définition de la classe **Atome()** commence par l'assignation de la variable **table**. Une variable définie à cet endroit fait partie de l'espace de noms de la classe. C'est donc un ***attribut de classe***, dans lequel nous plaçons une liste d'informations concernant les 10 premiers éléments du tableau périodique de Mendeléev.

Pour chacun de ces éléments, la liste contient un tuple : (nom de l'élément, nombre de neutrons), à l'indice qui correspond au numéro atomique. Comme il n'existe pas d'élément de numéro atomique zéro, nous avons placé à l'indice zéro dans la liste, l'objet spécial **None**. (A priori, nous aurions pu placer à cet endroit n'importe quelle autre valeur, puisque cet indice ne sera pas utilisé. L'objet None de Python nous semble cependant particulièrement explicite).

Viennent ensuite les définitions de deux méthodes :

- Le constructeur **__init__()** sert essentiellement ici à générer trois ***attributs d'instance***, destinés à mémoriser respectivement les nombres de protons, d'électrons et de neutrons pour chaque objet-atome construit à partir de cette classe (Les attributs d'instance sont des variables liées à **self**).
  Notez bien la technique utilisée pour obtenir le nombre de neutrons à partir de l'attribut de classe, en mentionnant le nom de la classe elle-même dans une qualification par points.
- La méthode **affiche()** utilise à la fois les attributs d'instance, pour retrouver les nombres de protons, d'électrons et de neutrons de l'objet courant, et l'attribut de classe (lequel est commun à tous les objets) pour en extraire le nom d'élément correspondant. Veuillez aussi remarquer au passage l'utilisation de la technique de formatage des chaînes (cfr. page 126).

La définition de la classe **Ion()** comporte des parenthèses. Il s'agit donc d'une ***classe dérivée***, sa ***classe parente*** étant bien entendu la classe **Atome()** qui précède.

Les méthodes de cette classe sont des variantes de celles de la classe atome. Elles devront donc vraisemblablement faire appel à celles-ci. Cette remarque est importante :

Comment peut-on, à l'intérieur de la définition d'une classe, faire appel à une méthode définie dans une autre classe ?

Il ne faut pas perdre de vue, en effet, qu'une méthode se rattache toujours à l'instance qui sera générée à partir de la classe (instance représentée par **self** dans la définition). Si une méthode doit faire appel à une autre méthode définie dans une autre classe, il faut pouvoir lui transmettre la référence de l'instance à laquelle elle doit s'associer. Comment faire ? C'est très simple :

Lorsque dans la définition d'une classe, on souhaite faire appel à une méthode définie dans une autre classe, on doit lui transmettre la référence de l'instance comme premier argument.

C'est ainsi que dans notre script, par exemple, la méthode **affiche()** de la classe **Ion()** peut faire appel à la méthode **affiche()** de la classe **Atome()** : les informations affichées seront bien celles de l'objet-ion courant, puisque sa référence a été transmise dans l'instruction d'appel :

```
Atome.affiche(self)
```

(dans cette instruction, **self** est bien entendu la référence de l'instance courante).

De la même manière (vous en verrez de nombreux autres exemples plus loin), la méthode constructeur de la classe **Ion()** fait appel à la méthode constructeur de sa classe parente, dans :

```
Atome.__init__(self, nat)
```

Résumé : Définition et utilisation d'une classe

```
####################################
# Programme Python type            #
# auteur : G.Swinnen, Liège, 2003  #
# licence : GPL                    #
####################################

class Point:
    """point mathématique"""
    def __init__(self, x, y):
        self.x = x
        self.y = y

class Rectangle:
    """rectangle"""
    def __init__(self, ang, lar, hau):
        self.ang = ang
        self.lar = lar
        self.hau = hau

    def trouveCentre(self):
        xc = self.ang.x + self.lar /2
        yc = self.ang.y + self.hau /2
        return Point(xc, yc)

class Carre(Rectangle):
    """carré = rectangle particulier"""
    def __init__(self, coin, cote):
        Rectangle.__init__(self,
                  coin, cote, cote)
        self.cote = cote

    def surface(self):
        return self.cote**2

###########################
## Programme principal : ##

# coord. de 2 coins sup. gauches :
csgR = Point(40,30)
csgC = Point(10,25)

# "boîtes" rectangulaire et carrée :
boiteR = Rectangle(csgR, 100, 50)
boiteC = Carre(csgC, 40)

# Coordonnées du centre pour chacune :
cR = boiteR.trouveCentre()
cC = boiteC.trouveCentre()

print "centre du rect. :", cR.x, cR.y
print "centre du carré :", cC.x, cC.y

print "surf. du carré :",
print boiteC.surface()
```

La classe est un moule servant à produire des objets. Chacun d'eux sera une instance de la classe considérée.

Les instances de la classe Point() seront des objets très simples qui posséderont seulement un attribut 'x' et un attribut 'y' ; ils ne seront dotés d'aucune méthode.

Le mot réservé self désigne toutes les instances qui seront produites par cette classe

Les instances de la classe Rectangle() posséderont 3 attributs : le premier ('ang') doit être lui-même un objet de classe Point(). Il servira à mémoriser les coordonnées de l'angle supérieur gauche du rectangle.

La classe Rectangle() comporte une méthode, qui renverra un objet de classe Point() au programme appelant.

Carre() est une classe dérivée, qui hérite les attributs et méthodes de la classe Rectangle(). Son constructeur doit faire appel au constructeur de la classe parente, en lui transmettant la référence de l'instance (self) comme premier argument.

La classe Carre() comporte une méthode de plus que sa classe parente.

Pour créer (ou instancier) un objet, il suffit d'affecter une classe à une variable. Les instructions ci-contre créent donc deux objets de la classe Point()...

... et celles-ci, encore deux autres objets. Note : par convention, le nom d'une classe commence par une lettre majuscule

La méthode trouveCentre() fonctionne pour les objets des deux types, puisque la classe Carre() a hérité de classe Rectangle().

Par contre, la méthode surface() ne peut être invoquée que pour les objets carrés.

12.6 Modules contenant des bibliothèques de classes

Vous connaissez déjà depuis longtemps l'utilité des modules Python. Vous savez qu'ils servent à regrouper des bibliothèques de classes et de fonctions. A titre d'exercice de révision, vous allez créer vous-même un nouveau module de classes, en encodant les lignes d'instruction ci-dessous dans un fichier que vous nommerez **formes.py** :

```
class Rectangle:
    "Classe de rectangles"
    def __init__(self, longueur =30, largeur =15):
        self.L = longueur
        self.l = largeur
        self.nom ="rectangle"

    def perimetre(self):
        return "(%s + %s) * 2 = %s" % (self.L, self.l,
                                            (self.L + self.l)*2)
    def surface(self):
        return "%s * %s = %s" % (self.L, self.l, self.L*self.l)

    def mesures(self):
        print "Un %s de %s sur %s" % (self.nom, self.L, self.l)
        print "a une surface de %s" % (self.surface(),)
        print "et un périmètre de %s\n" % (self.perimetre(),)

class Carre(Rectangle):
    "Classe de carrés"
    def __init__(self, cote =10):
        Rectangle.__init__(self, cote, cote)
        self.nom ="carré"

if __name__ == "__main__":
    r1 = Rectangle(15, 30)
    r1.mesures()
    c1 = Carre(13)
    c1.mesures()
```

Une fois ce module enregistré, vous pouvez l'utiliser de deux manières : Soit vous en lancez l'exécution comme celle d'un programme ordinaire, soit vous l'importez dans un script quelconque ou depuis la ligne de commande, pour en utiliser les classes :

```
>>> import formes
>>> f1 = formes.Rectangle(27, 12)
>>> f1.mesures()
Un rectangle de 27 sur 12
a une surface de 27 * 12 = 324
et un périmètre de (27 + 12) * 2 = 78

>>> f2 = formes.Carre(13)
>>> f2.mesures()
Un carré de 13 sur 13
a une surface de 13 * 13 = 169
et un périmètre de (13 + 13) * 2 = 52
```

On voit dans ce script que la classe **Carre()** est construite par dérivation à partir de la classe **Rectangle()** dont elle hérite toutes les caractéristiques. En d'autres termes, la classe **Carre()** est une classe fille de la classe **Rectangle()**.

Vous pouvez remarquer encore une fois que le constructeur de la classe **Carre()** fait appel au constructeur de sa classe parente (**Rectangle.__init__()**), en lui transmettant la référence de l'instance (c.à.d. **self**) comme premier argument.

Quant à l'instruction :

```
if __name__ == "__main__":
```

placée à la fin du module, elle sert à déterminer si le module est "lancé" en tant que programme (auquel cas les instructions qui suivent doivent être exécutées), ou au contraire utilisé comme une bibliothèque de classes importée ailleurs. Dans ce cas cette partie du code est sans effet.

Exercices :

e 157. Définissez une classe **Cercle()**. Les objets construits à partir de cette classe seront des cercles de tailles variées. En plus de la méthode constructeur (qui utilisera donc un paramètre **rayon**), vous définirez une méthode **surface()**, qui devra renvoyer la surface du cercle.

Définissez ensuite une classe **Cylindre()** dérivée de la précédente. Le constructeur de cette nouvelle classe comportera les deux paramètres **rayon** et **hauteur**. Vous y ajouterez une méthode **volume()** qui devra renvoyer le volume du cylindre.
(Rappel : Volume d'un cylindre = surface de section x hauteur).

Exemple d'utilisation de cette classe :

```
>>> cyl = Cylindre(5, 7)
>>> print cyl.surface()
78.54
>>> print cyl.volume()
549.78
```

e 158. Complétez l'exercice précédent en lui ajoutant encore une classe **Cone()**, qui devra dériver cette fois de la classe **Cylindre()**, et dont le constructeur comportera lui aussi les deux paramètres **rayon** et **hauteur**. Cette nouvelle classe possédera sa propre méthode **volume()**, laquelle devra renvoyer le volume du cône.
(Rappel : Volume d'un cône = volume du cylindre correspondant divisé par 3).

Exemple d'utilisation de cette classe :

```
>>> co = Cone(5,7)
>>> print co.volume()
183.26
```

e 159. Définissez une classe **JeuDeCartes()** permettant d'instancier des objets "jeu de cartes" dont le comportement soit similaire à celui d'un vrai jeu de cartes. La classe devra comporter au moins les trois méthodes suivantes :

- méthode constructeur : création et remplissage d'une liste de 52 éléments, qui sont eux-mêmes des tuples de 2 éléments contenant les caractéristiques de chacune des 52 cartes.
Pour chacune d'elles, il faut en effet mémoriser séparément un nombre entier indiquant la valeur (2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, les 4 dernières valeurs étant celles des valet, dame, roi et as), et un autre nombre entier indiquant la couleur de la carte (c.à.d. 3,2,1,0 pour Coeur, Carreau, Trèfle & Pique).
Dans une telle liste, l'élément (11,2) désigne donc le valet de Trèfle, et la liste terminée doit être du type : [(2, 0), (3,0), (3,0), (4,0), ... (12,3), (13,3), (14,3)]

- méthode **nom_carte()** : cette méthode renvoie sous la forme d'une chaîne l'identité d'une carte quelconque, dont on lui a fourni le tuple descripteur en argument.
Par exemple, l'instruction :
`print jeu.nom_carte((14, 3))` doit provoquer l'affichage de : `As de pique`

- méthode **battre()** : comme chacun sait, battre les cartes consiste à les mélanger.
Cette méthode sert donc à mélanger les éléments de la liste contenant les cartes, quel qu'en soit le nombre.

- méthode **tirer()** : lorsque cette méthode est invoquée, une carte ***est retirée*** du jeu. Le tuple contenant sa valeur et sa couleur est renvoyé au programme appelant. On retire toujours la première carte de la liste. Si cette méthode est invoquée alors qu'il ne reste plus aucune carte dans la liste, il faut alors renvoyer l'objet spécial None au programme appelant.

Exemple d'utilisation de la classe **JeuDeCartes()** :

```
jeu = JeuDeCartes()              # instanciation d'un objet
jeu.battre()                     # mélange des cartes
for n in range(53):              # tirage des 52 cartes :
   c = jeu.tirer()
   if c == None:                 # il ne reste plus aucune carte
      print 'Terminé !'          # dans la liste
   else:
      print jeu.nom_carte(c)     # valeur et couleur de la carte
```

e 160. Complément de l'exercice précédent : Définir deux joueurs A et B. Instancier deux jeux de cartes (un pour chaque joueur) et les mélanger. Ensuite, à l'aide d'une boucle, tirer 52 fois une carte de chacun des deux jeux et comparer leurs valeurs. Si c'est la première des 2 qui a la valeur la plus élevée, on ajoute un point au joueur A. Si la situation contraire se présente, on ajoute un point au joueur B. Si les deux valeurs sont égales, on passe au tirage suivant. Au terme de la boucle, comparer les comptes de A et B pour déterminer le gagnant.

Chapitre 13 : Classes & Interfaces graphiques

La programmation orientée objet convient particulièrement bien au développement d'applications avec interface graphique. Des bibliothèques de classes comme *Tkinter* ou *wxPython* fournissent une base de *widgets* très étoffée, que nous pouvons adapter à nos besoins par dérivation. Dans ce chapitre, nous allons utiliser à nouveau la bibliothèque *Tkinter*, mais en appliquant les concepts décrits dans les pages précédentes, et en nous efforçant de mettre en évidence les avantages qu'apporte ***l'orientation objet*** dans nos programmes.

13.1 "Code des couleurs" : un petit projet bien encapsulé

Nous allons commencer par un petit projet qui nous a été inspiré par le cours d'initiation à l'électronique. L'application que nous décrivons ci-après permet de retrouver rapidement le code de trois couleurs qui correspond à une résistance électrique de valeur bien déterminée.

Pour rappel, la fonction des résistances électriques consiste à s'opposer (à résister) plus ou moins bien au passage du courant. Les résistances se présentent concrètement sous la forme de petites pièces tubulaires cerclées de bandes de couleur (en général 3). Ces bandes de couleur indiquent la valeur numérique de la résistance, en fonction du code suivant :

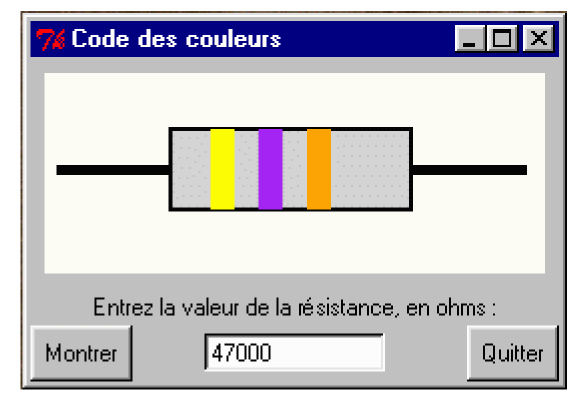
Chaque couleur correspond conventionnellement à l'un des chiffres de zéro à neuf :
Noir = 0 ; Brun = 1 ; Rouge = 2 ; Orange = 3 ; Jaune = 4 ;
Vert = 5 ; Bleu = 6 ; Violet = 7 ; Gris = 8 ; Blanc = 9.

On oriente la résistance de manière telle que les bandes colorées soient placées à gauche. La valeur de la résistance – exprimée en ohms (Ω) - s'obtient en lisant ces bandes colorées également à partir de la gauche : les deux premières bandes indiquent les deux premiers chiffres de la valeur numérique ; il faut ensuite accoler à ces deux chiffres un nombre de zéros égal à l'indication fournie par la troisième bande. Exemple concret :
Supposons qu'à partir de la gauche, les bandes colorées soient jaune, violette et verte.
La valeur de cette résistance est 4700000 Ω, ou 4700 kΩ, ou encore 4,7 MΩ..

Ce système ne permet évidemment de préciser une valeur numérique qu'avec deux chiffres significatifs seulement. Il est toutefois considéré comme largement suffisant pour la plupart des applications électroniques "ordinaires" (radio, TV, etc.)

a) Cahier des charges de notre programme :

Notre application doit faire apparaître une fenêtre comportant un dessin de la résistance, ainsi qu'un champ d'entrée dans lequel l'utilisateur peut encoder une valeur numérique. Un bouton "Montrer" déclenche la modification du dessin de la résistance, de telle façon que les trois bandes de couleur se mettent en accord avec la valeur numérique introduite.



Contrainte : Le programme doit accepter toute entrée numérique fournie sous forme entière ou réelle, dans les limites de 10 à 10^{11} Ω. Par exemple, une valeur telle que 4.78e6 doit être acceptée et arrondie correctement, c.à.d. convertie en 4800000 Ω.

b) Mise en oeuvre concrète

Nous construisons cette application simple sous la forme d'une ***classe***. Sa seule utilité présente consiste à nous fournir ***un espace de noms commun*** dans lequel nous pouvons ***encapsuler*** nos variables et nos fonctions, ce qui nous permet de nous passer de variables globales. En effet :

- Les variables auxquelles nous souhaitons pouvoir accéder de partout sont déclarées comme des attributs d'instance (nous attachons chacune d'elles à l'instance à l'aide de **self**).
- Les fonctions sont déclarées comme des méthodes, et donc attachées elles aussi à **self**.

Au niveau principal du programme, nous nous contentons d'instancier un objet de la classe ainsi construite (aucune méthode de cet objet n'est activée de l'extérieur).

```
class Application:
    def __init__(self):
        """Constructeur de la fenêtre principale"""
        self.root =Tk()
        self.root.title('Code des couleurs')
        self.dessineResistance()
        Label(self.root,
              text ="Entrez la valeur de la résistance, en ohms :").grid(row =2)
        Button(self.root, text ='Montrer',
               command =self.changeCouleurs).grid(row =3, sticky = W)
        Button(self.root, text ='Quitter',
               command =self.root.quit).grid(row =3, sticky = E)
        self.entree = Entry(self.root, width =14)
        self.entree.grid(row =3)
        # Code des couleurs pour les valeurs de zéro à neuf :
        self.cc = ['black','brown','red','orange','yellow',
                   'green','blue','purple','grey','white']
        self.root.mainloop()

    def dessineResistance(self):
        """Canevas avec un modèle de résistance à trois lignes colorées"""
        self.can = Canvas(self.root, width=250, height =100, bg ='ivory')
        self.can.grid(row =1, pady =5, padx =5)
        self.can.create_line(10, 50, 240, 50, width =5)         # fils
        self.can.create_rectangle(65, 30, 185, 70, fill ='light grey', width =2)
        # Dessin des trois lignes colorées (noires au départ) :
        self.ligne =[]              # on mémorisera les trois lignes dans 1 liste
        for x in range(85,150,24):
            self.ligne.append(self.can.create_rectangle(x,30,x+12,70,
                                                        fill='black',width=0))

    def changeCouleurs(self):
        """Affichage des couleurs correspondant à la valeur entrée"""
        self.v1ch = self.entree.get()       # la méthode get() renvoie une chaîne
        try:
            v = float(self.v1ch)            # conversion en valeur numérique
        except:
            err =1                          # erreur : entrée non numérique
        else:
            err =0
        if err ==1 or v < 10 or v > 1e11 :
            self.signaleErreur()            # entrée incorrecte ou hors limites
        else:
            li =[0]*3                       # liste des 3 codes à afficher
            logv = int(log10(v))            # partie entière du logarithme
            ordgr = 10**logv                # ordre de grandeur
            # extraction du premier chiffre significatif :
            li[0] = int(v/ordgr)            # partie entière
            decim = v/ordgr - li[0]         # partie décimale
            # extraction du second chiffre significatif :
            li[1] = int(decim*10 +.5)           # +.5 pour arrondir correctement
            # nombre de zéros à accoler aux 2 chiffres significatifs :
            li[2] = logv -1
            # Coloration des 3 lignes :
```

```
55.             for n in range(3):
56.                 self.can.itemconfigure(self.ligne[n], fill =self.cc[li[n]])
57.
58.     def signaleErreur(self):
59.         self.entree.configure(bg ='red')            # colorer le fond du champ
60.         self.root.after(1000, self.videEntree)      # après 1 seconde, effacer
61.
62.     def videEntree(self):
63.         self.entree.configure(bg ='white')          # rétablir le fond blanc
64.         self.entree.delete(0, len(self.v1ch))       # enlever les car. présents
65.
66. # Programme principal :
67. from Tkinter import *
68. from math import log10                    # logarithmes en base 10
69. f = Application()                         # instanciation de l'objet application
```

Commentaires :

- Ligne 1 : La classe est définie sans référence à une classe parente (pas de parenthèses). Il s'agira donc d'une nouvelle classe indépendante.
- Lignes 2 à 14 : Le constructeur de la classe instancie les *widgets* nécessaires : pour améliorer la lisibilité du programme, on a placé l'instanciation du canevas (avec le dessin de la résistance) dans une méthode séparée **dessineResistance()**. Les boutons et le libellé ne sont pas mémorisés dans des variables, parce que l'on ne souhaite pas y faire référence ailleurs dans le programme.
  Le positionnement des *widgets* dans la fenêtre utilise la méthode **grid()**, décrite à la page 92.
- Lignes 15-17 : Le code des couleurs est mémorisé dans une simple liste.
- Ligne 18 : La dernière instruction du constructeur démarre l'application.
- Lignes 20 à 30 : Le dessin de la résistance se compose d'une ligne et d'un premier rectangle gris clair, pour le corps de la résistance et ses deux fils. Trois autres rectangles figureront les bandes colorées que le programme devra modifier en fonction des entrées de l'utilisateur. Ces bandes sont noires au départ ; elles sont référencées dans la liste **self.ligne**.
- Lignes 32 à 53 : Ces lignes contiennent l'essentiel de la fonctionnalité du programme.
  L'entrée brute fournie par l'utilisateur est acceptée sous la forme d'une chaîne de caractères.
  A la ligne 36, on essaie de convertir cette chaîne en une valeur numérique de type *float*. Si la conversion échoue, on mémorise l'erreur. Si l'on dispose bien d'une valeur numérique, on vérifie ensuite qu'elle se situe effectivement dans l'intervalle autorisé (de 10 Ω à 10^{11} Ω). Si une erreur est détectée, on signale à l'utilisateur que son entrée est incorrecte en colorant de rouge le fond du champ d'entrée, qui est ensuite vidé de son contenu (lignes 55 à 61).
- Lignes 45-46 : Les mathématiques viennent à notre secours pour extraire de la valeur numérique son ordre de grandeur (c.à.d. l'exposant de 10 le plus proche). Veuillez consulter votre cours de mathématiques pour de plus amples explications concernant les logarithmes.
- Lignes 47-48 : Une fois connu l'ordre de grandeur, il devient relativement facile d'extraire du nombre traité ses deux premiers chiffres significatifs. Exemple : Supposons que la valeur entrée soit 31687. Le logarithme de ce nombre est 4,50088... dont la partie entière (4) nous donne l'ordre de grandeur de la valeur entrée (soit 10^4). Pour extraire de celle-ci son premier chiffre significatif, il suffit de la diviser par 10^4, soit 10000, et de conserver seulement la partie entière du résultat (3).

- Lignes 49 à 51 : Le résultat de la division effectuée dans le paragraphe précédent est 3,1687. Nous récupérons la partie décimale de ce nombre à la ligne 49, soit 0,1687 dans notre exemple. Si nous le multiplions par dix, ce nouveau résultat comporte une partie entière qui n'est rien d'autre que notre second chiffre significatif (1 dans notre exemple).
  Nous pourrions facilement extraire ce dernier chiffre, mais puisque c'est le dernier, nous souhaitons encore qu'il soit correctement arrondi. Pour ce faire, il suffit d'ajouter une demi unité au produit de la multiplication par dix, avant d'en extraire la valeur entière. Dans notre exemple, en effet, ce calcul donnera donc 1,687 + 0,5 = 2,187 , dont la partie entière (2) est bien la valeur arrondie recherchée.
- Ligne 53 : Le nombre de zéros à accoler aux deux chiffres significatifs correspond au calcul de l'ordre de grandeur. Il suffit de retirer une unité au logarithme.
- Ligne 56 : Pour attribuer une nouvelle couleur à un objet déjà dessiné dans un canevas, on utilise la méthode **itemconfigure()**. Nous utilisons donc cette méthode pour modifier l'option **fill** de chacune des bandes colorées, en utilisant les noms de couleur extraits de la liste **self.cc** grâce à aux trois indices **li[1]**, **li[2]** et **li[3]** qui contiennent les 3 chiffres correspondants.

c) Exercices :

e 161. Modifiez le script ci-dessus de telle manière que le fond d'image devienne bleu clair ('*light blue*'), que le corps de la résistance devienne beige ('*beige*'), que le fil de cette résistance soit plus fin, et que les bandes colorées indiquant la valeur soient plus larges.

e 162. Modifiez le script ci-dessus de telle manière que l'image dessinée soit deux fois plus grande.

e 163. Modifiez le script ci-dessus de telle manière qu'il devienne possible d'entrer aussi des valeurs de résistances comprises entre 1 et 10 Ω. Pour ces valeurs, le premier anneau coloré devra rester noir, les deux autres indiqueront la valeur en Ω et dixièmes d' Ω.

e 164. Modifiez le script ci-dessus de telle façon que le bouton "Montrer" ne soit plus nécessaire. Dans votre script modifié, il suffira de frapper <Enter> après avoir entré la valeur de la résistance, pour que l'affichage s'active.

e 165. Modifiez le script ci-dessus de telle manière que les trois bandes colorées redeviennent noires dans les cas où l'utilisateur fournit une entrée inacceptable.

13.2 "OscilloGraphe" : un widget personnalisé construit par dérivation

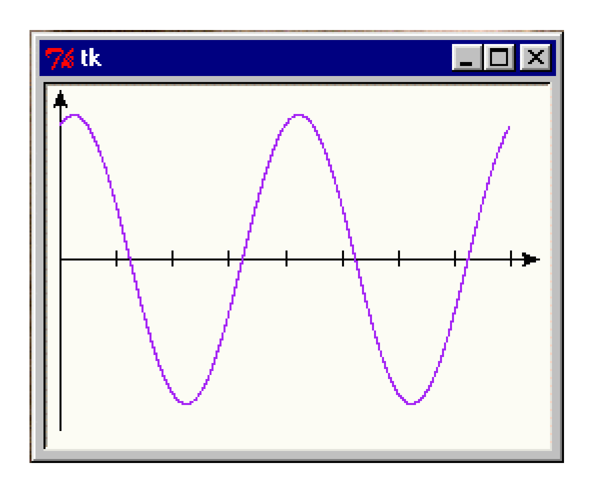
Dans l'exercice précédent, nous n'avons exploité qu'une seule caractéristique des classes : l'***encapsulation***. Celle-ci nous a permis d'écrire un programme dans lequel les différentes fonctions (qui sont donc devenues des ***méthodes***) peuvent chacune accéder à un même *pool* de variables : toutes celles qui sont définies comme étant attachées à **self**. Toutes ces variables peuvent être considérées en quelque sorte comme des variables globales, à l'intérieur de l'objet.

Comprenez bien toutefois qu'il ne s'agit pas de véritables variables globales. Elles restent en effet strictement confinées à l'intérieur de l'objet, et il est déconseillé de vouloir y accéder de l'extérieur[48]. D'autre part, tous les objets que vous instancierez à partir d'une même classe posséderont chacun leur propre jeu de ces variables, qui sont donc bel et bien ***encapsulées*** dans ces objets. On les appelle pour cette raison des ***attributs d'instance***.

Le projet qui suit va nous entraîner encore un peu plus loin. Nous allons en effet y construire une nouvelle classe de *widget*, qu'il sera possible d'intégrer dans nos projets futurs comme n'importe quel *widget* standard. Cette nouvelle classe sera construite par ***dérivation*** d'une classe existante, afin d'illustrer au passage le mécanisme d'***héritage***.

Le sujet concret de cette application nous est inspiré par le cours de physique. Pour rappel :

Un mouvement vibratoire harmonique *se définit comme étant la projection d'un mouvement circulaire uniforme sur une droite. Les positions successives d'un mobile qui effectue ce type de mouvement sont traditionnellement repérées par rapport à une position centrale : on les appelle alors des* ***élongations****. L'équation qui décrit l'évolution de l'élongation d'un tel mobile au cours du temps est toujours de la forme* $e = A\sin(2\pi f t + \varphi)$ *, dans laquelle* ***e*** *représente l'élongation du mobile à tout instant* ***t*** *. Les constantes* ***A****,* ***f*** *et* **φ** *désignent respectivement l'amplitude, la fréquence et la phase du mouvement vibratoire.*



Le but du présent projet est de fournir un instrument de visualisation simple de ces différents concepts, à savoir un système d'affichage automatique de graphiques élongation/temps. L'utilisateur pourra choisir librement les valeurs des paramètres **A**, **f** et **φ** , et observer les courbes qui en résultent.

Le *widget* que nous allons construire d'abord s'occupera de l'affichage proprement dit. Nous construirons ensuite d'autres *widgets* pour faciliter l'entrée des paramètres **A**, **f** et **φ** .

Veuillez donc encoder le script ci-dessous et le sauvegarder dans un fichier, auquel vous donnerez le nom **oscillo.py** . Vous réaliserez ainsi un véritable ***module*** contenant une classe (vous pourrez par la suite ajouter d'autres classes dans ce même module, si le coeur vous en dit).

48 Comme nous l'avons déjà signalé précédemment, Python vous permet d'accéder aux attributs d'instance en utilisant la qualification des noms par points. D'autres langages de programmation l'interdisent, ou bien ne l'autorisent que moyennant une déclaration particulière de ces attributs (distinction entre attributs privés et publics).
Sachez en tous cas que ce n'est pas recommandé : le bon usage de la programmation orientée objet stipule en effet que vous ne devez pouvoir accéder aux attributs des objets que par l'intermédiaire de méthodes spécifiques.

```
1.   from Tkinter import *
2.   from math import sin, pi
3.
4.   class OscilloGraphe(Canvas):
5.       "Canevas spécialisé, pour dessiner des courbes élongation/temps"
6.       def __init__(self, boss =None, larg=200, haut=150):
7.           "Constructeur du graphique : axes et échelle horiz."
8.           # construction du widget parent :
9.           Canvas.__init__(self)                          # appel au constructeur
10.          self.configure(width=larg, height=haut)        # de la classe parente
11.          self.larg, self.haut = larg, haut                       # mémorisation
12.          # tracé des axes de référence :
13.          self.create_line(10, haut/2, larg, haut/2, arrow=LAST)   # axe X
14.          self.create_line(10, haut-5, 10, 5, arrow=LAST)          # axe Y
15.          # tracé d'une échelle avec 8 graduations :
16.          pas = (larg-25)/8.          # intervalles de l'échelle horizontale
17.          for t in range(1, 9):
18.              stx = 10 + t*pas        # +10 pour partir de l'origine
19.              self.create_line(stx, haut/2-4, stx, haut/2+4)
20.
21.      def traceCourbe(self, freq=1, phase=0, ampl=10, coul='red'):
22.          "tracé d'un graphique élongation/temps sur 1 seconde"
23.          curve =[]                       # liste des coordonnées
24.          pas = (self.larg-25)/1000.      # l'échelle X correspond à 1 seconde
25.          for t in range(0,1001,5):       # que l'on divise en 1000 ms.
26.              e = ampl*sin(2*pi*freq*t/1000 - phase)
27.              x = 10 + t*pas
28.              y = self.haut/2 - e*self.haut/25
29.              curve.append((x,y))
30.          n = self.create_line(curve, fill=coul, smooth=1)
31.          return n                        # n = numéro d'ordre du tracé
32.
33.  #### Code pour tester la classe : ####
34.
35.  if __name__ == '__main__':
36.      root = Tk()
37.      gra = OscilloGraphe(root, 250, 180)
38.      gra.pack()
39.      gra.configure(bg ='ivory', bd =2, relief=SUNKEN)
40.      gra.traceCourbe(2, 1.2, 10, 'purple')
41.      root.mainloop()
```

Le niveau principal du script est constitué par les lignes 35 à 41. Comme nous l'avons déjà expliqué à la page 164, les lignes de code situées après l'instruction **if __name__ == '__main__':** ne sont pas exécutées si le script est importé en tant que module. Si on lance le script comme application principale, par contre, ces instructions sont exécutées.

Nous disposons ainsi d'un mécanisme intéressant, qui nous permet d'intégrer des instructions de test à l'intérieur des modules, même si ceux-ci sont destinés à être importés dans d'autres scripts.

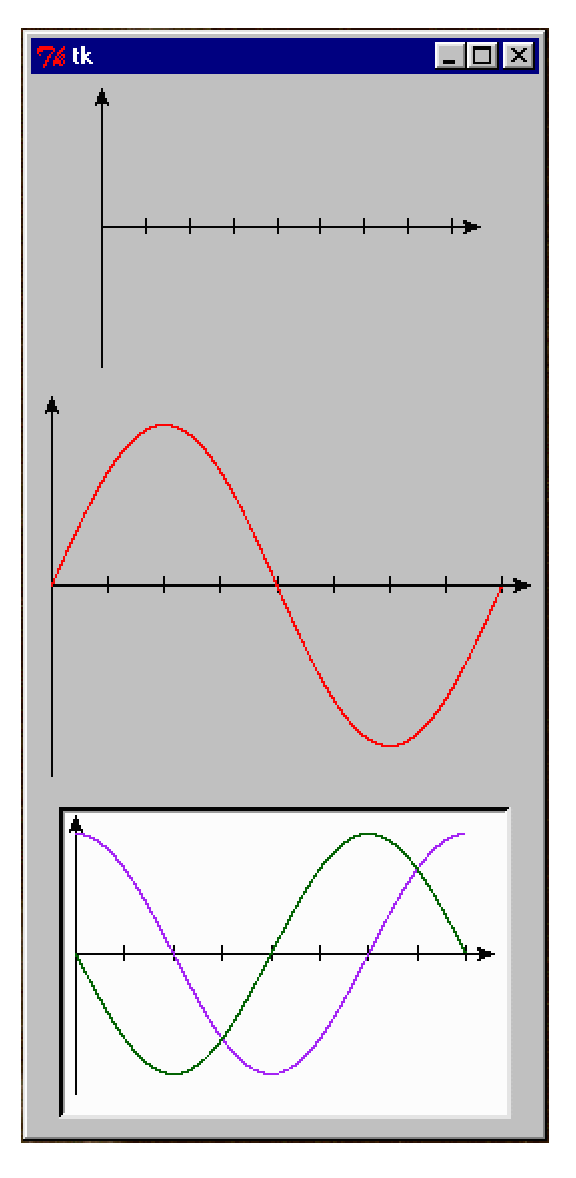
Lancez donc l'exécution du script de la manière habituelle. Vous devriez obtenir un affichage similaire à celui qui est reproduit à la page précédente.

Expérimentation :

Nous commenterons les lignes importantes du script un peu plus loin dans ce texte. Mais commençons d'abord par expérimenter quelque peu la classe que nous venons de construire.

Ouvrez une fenêtre de terminal (*"Python shell"*), et entrez les instructions ci-dessous directement à la ligne de commande :

```
>>> from oscillo import *
>>> g1 = OscilloGraphe()
>>> g1.pack()
```



Après importation des classes du module **oscillo**, nous instancions un premier objet **g1** , de la classe **OscilloGraphe()**.

Puisque nous ne fournissons aucun argument, l'objet possède les dimensions par défaut, définies dans le constructeur de la classe. Remarquons au passage que nous n'avons même pas pris la peine de définir d'abord une fenêtre maître pour y placer ensuite notre widget. *Tkinter* nous pardonne cet oubli et nous en fournit une automatiquement !

```
>>> g2 = OscilloGraphe(haut=200, larg=250)
>>> g2.pack()
>>> g2.traceCourbe()
```

Par ces instructions, nous créons un second widget de la même classe, en précisant cette fois ses dimensions (hauteur et largeur, dans n'importe quel ordre).

Ensuite, nous activons la méthode **traceCourbe()** associée à ce widget. Étant donné que nous ne lui fournissons aucun argument, la sinusoïde qui apparaît correspond aux valeurs prévues par défaut pour les paramètres **A**, **f** et **φ** .

```
>>> g3 = OscilloGraphe(larg=220)
>>> g3.configure(bg='white', bd=3, relief=SUNKEN)
>>> g3.pack(padx=5,pady=5)
>>> g3.traceCourbe(phase=1.57, coul='purple')
>>> g3.traceCourbe(phase=3.14, coul='dark green')
```

Pour comprendre la configuration de ce troisième widget, il faut nous rappeler que la classe **OscilloGraphe()** a été construite par dérivation de la classe **Canvas()**. Elle hérite donc de toutes les propriétés de celle-ci, ce qui nous permet de choisir la couleur de fond, la bordure, etc., en utilisant les mêmes arguments que ceux qui sont à notre disposition lorsque nous configurons un canevas.

Nous faisons ensuite apparaître deux tracés successifs, en faisant appel deux fois à la méthode **traceCourbe()**, à laquelle nous fournissons des arguments pour la phase et la couleur.

Exercice :

e 166. Créez un quatrième widget, de taille 400 x 300, couleur de fond jaune, et faites-y apparaître plusieurs courbes correspondant à des fréquences et des amplitudes différentes.

Il est temps à présent que nous analysions la structure de la classe qui nous a permis d'instancier tous ces *widgets*. Nous avons enregistré cette classe dans le module **oscillo.py** (voir page 171).

a) Cahier des charges :

Nous souhaitons définir une nouvelle classe de *widget*, capable d'afficher automatiquement les graphiques élongation/temps correspondant à divers mouvements vibratoires harmoniques.

Ce *widget* doit pouvoir être dimensionné à volonté au moment de son instanciation. Il fait apparaître deux axes cartésiens **X** et **Y** munis de flèches. L'axe **X** représente l'écoulement du temps pendant une seconde au total, et il est muni d'une échelle comportant 8 intervalles.

Une méthode **traceCourbe()** est associée à ce *widget*. Elle provoque le tracé du graphique élongation/temps pour un mouvement vibratoire dont on fournit la fréquence (entre 0.25 et 10 Hz), la phase (entre 0 et 2π radians) et l'amplitude (entre 1 et 10 ; échelle arbitraire).

b) Implémentation :

- Ligne 4 : La classe **OscilloGraphe()** est créée par dérivation de la classe **Canvas()**.
  Elle hérite donc de toutes les propriétés de celle-ci : on pourra configurer les objets de cette nouvelle classe en utilisant les nombreuses options déjà disponibles pour la classe **Canvas()**.
- Ligne 6 : La méthode "constructeur" utilise 3 paramètres, qui sont tous optionnels puisque chacun d'entre eux possède une valeur par défaut. Le paramètre **boss** ne sert qu'à réceptionner la référence d'une fenêtre maîtresse éventuelle (voir exemples suivants). Les paramètres **larg** et **haut** (largeur et hauteur) servent à assigner des valeurs aux options **width** et **height** du canevas parent, au moment de l'instanciation.
- Lignes 9 & 10 : La première opération que doit accomplir le constructeur d'une classe dérivée, c'est activer le constructeur de sa classe parente. En effet : nous ne pouvons hériter de toute la fonctionnalité de la classe parente, que si cette fonctionnalité a été effectivement mise en place.
  Nous activons donc le constructeur de la classe **Canvas()** à la ligne 9 , et nous ajustons deux de ses options à la ligne 10. Notez au passage que nous pourrions condenser ces deux lignes en une seule, qui deviendrait en l'occurrence :
  ```
  Canvas.__init__(self, width=larg, height=haut)
  ```
  Rappel : comme cela a été expliqué à la page 161, nous devons transmettre à ce constructeur la référence de l'instance présente (self) comme premier argument.
- Ligne 11 : Il est nécessaire de mémoriser les paramètres **larg** et **haut** dans des variables d'instance, parce que nous devrons pouvoir y accéder aussi dans la méthode **traceCourbe()**.
- Lignes 13 & 14 : Pour tracer les axes X et Y, nous utilisons les paramètres **larg** et **haut**, ainsi ces axes sont automatiquement mis à dimension. L'option **arrow=LAST** permet de faire apparaître une petite flèche à l'extrémité de chaque ligne.
- Lignes 16 à 19 : Pour tracer l'échelle horizontale, on commence par réduire de 25 pixels la largeur disponible, de manière à ménager des espaces aux deux extrémités. On divise ensuite en 8 intervalles, que l'on visualise sous la forme de 8 petits traits verticaux.
- Ligne 21 : La méthode **traceCourbe()** pourra être invoquée avec quatre arguments.
  Chacun d'entre eux pourra éventuellement être omis, puisque chacun des paramètres correspondants possède une valeur par défaut. Il sera également possible de fournir les arguments dans n'importe quel ordre, comme nous l'avons déjà expliqué à la page 76.

- Lignes 23 à 31 : Pour le tracé de la courbe, la variable **t** prend successivement toutes les valeurs de 0 à 1000, et on calcule à chaque fois l'élongation **e** correspondante, à l'aide de la formule théorique (ligne 26). Les couples de valeurs **t** & **e** ainsi trouvées sont mises à l'échelle et transformées en coordonnées x, y aux lignes 27 & 28, puis accumulées dans la liste **curve**.
- Lignes 30 & 31 : La méthode **create_line()** trace alors la courbe correspondante en une seule opération, et elle retourne le numéro d'ordre du nouvel objet ainsi instancié dans le canevas (ce numéro d'ordre nous permettra d'y accéder encore par après : pour l'effacer, par exemple). L'option **smooth =1** améliore l'aspect final, par lissage.

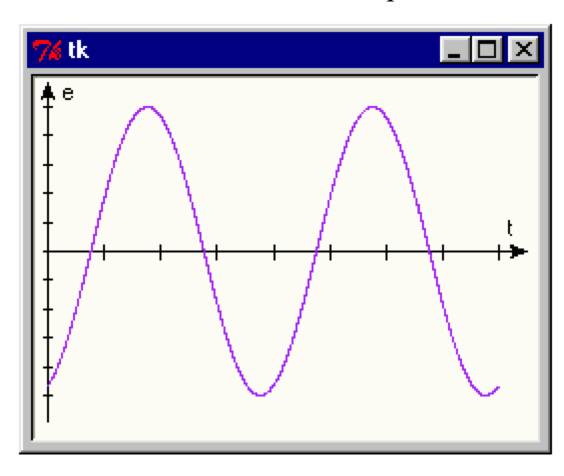
Exercices :

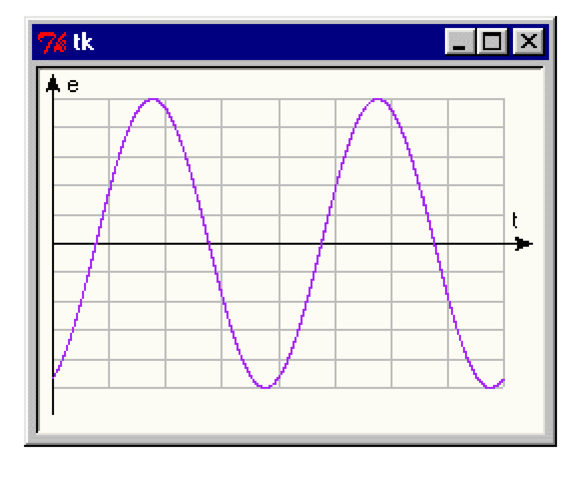
e 167. Modifiez le script de manière à ce que l'axe de référence vertical comporte lui aussi une échelle, avec 5 tirets de part et d'autre de l'origine.

e 168. Comme les *widgets* de la classe **Canvas()** dont il dérive, votre *widget* peut intégrer des indications textuelles. Il suffit pour cela d'utiliser la méthode **create_text()**, comme nous l'avons déjà fait dans l'exercice optionnel des pages et suivantes. Cette méthode attend au moins trois arguments : les coordonnées **x** et **y** de l'emplacement où vous voulez faire apparaître votre texte, et puis le texte lui-même, bien entendu. D'autres arguments peuvent être transmis sous forme d'options, pour préciser par exemple la police de caractères et sa taille. Afin de voir comment cela fonctionne, ajoutez provisoirement la ligne suivante dans le constructeur de la classe **OscilloGraphe()**, puis relancez le script :

```
self.create_text(130, 30, text = "Essai", anchor =CENTER)
```

Utilisez cette méthode pour ajouter au *widget* les indications suivantes aux extrémités des axes de référence : **e** (*pour "élongation"*) le long de l'axe vertical, et **t** (*pour "temps"*) le long de l'axe horizontal. Le résultat pourrait ressembler à ceci (figure de gauche) :





e 169. Vous pouvez compléter encore votre *widget*, en y faisant apparaître une grille de référence, plutôt que de simples tirets le long des axes. Pour éviter que cette grille ne soit trop visible, vous pouvez colorer ses traits en gris (option **fill = 'grey'**), comme dans la figure de droite.

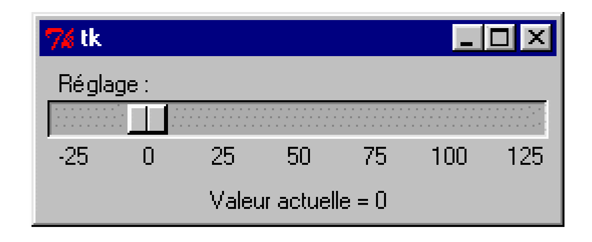
e 170. Complétez encore votre *widget* en y faisant apparaître des repères numériques.

13.3 "Curseurs" : un widget composite

Dans l'exercice précédent, vous avez construit un nouveau type de *widget* que vous avez sauvegardé dans le module **oscillo.py**. Conservez soigneusement ce module, car vous l'intégrerez bientôt dans un projet plus complexe.

Pour l'instant, vous allez construire encore un autre *widget*, plus interactif cette fois. Il s'agira d'une sorte de panneau de contrôle comportant trois curseurs de réglage et une case à cocher. Comme le précédent, ce *widget* est destiné à être réutilisé dans une application de synthèse.

13.3.1 Présentation du widget "Scale"



Commençons d'abord par découvrir un *widget* de base, que nous n'avions pas encore utilisé jusqu'ici :
Le *widget* **Scale** se présente comme un curseur qui coulisse devant une échelle. Il permet à l'utilisateur de choisir rapidement la valeur d'un paramètre quelconque, d'une manière très attrayante.

Le petit script ci-dessous vous montre comment le paramétrer et l'utiliser dans une fenêtre :

```
from Tkinter import *

def updateLabel(x):
    lab.configure(text='Valeur actuelle = ' + str(x))

root = Tk()
Scale(root, length=250, orient=HORIZONTAL, label ='Réglage :',
      troughcolor ='dark grey', sliderlength =20,
      showvalue =0, from_=-25, to=125, tickinterval =25,
      command=updateLabel).pack()
lab = Label(root)
lab.pack()

root.mainloop()
```

Ces lignes ne nécessitent guère de commentaires.
Vous pouvez créer des widgets **Scale** de n'importe quelle taille (option **length**), en orientation horizontale (comme dans notre exemple) ou verticale (option **orient** = **VERTICAL**).
Les options **from_** (attention : n'oubliez pas le caractère 'souligné' !) et **to** définissent la plage de réglage. L'intervalle entre les repères numériques est défini dans l'option **tickinterval**, etc.

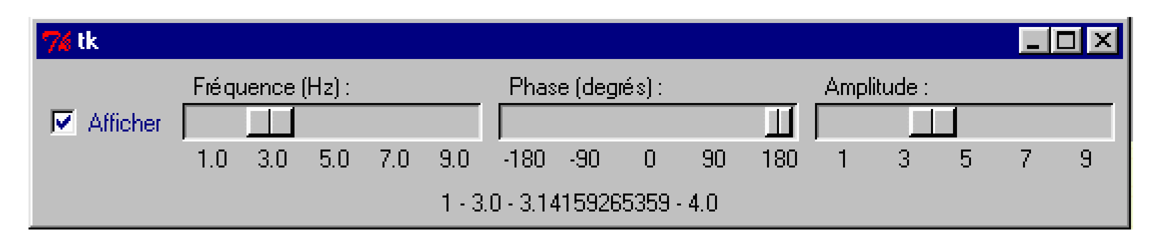
La fonction désignée dans l'option **command** est appelée automatiquement chaque fois que le curseur est déplacé, et la position actuelle du curseur par rapport à l'échelle lui est transmise en argument. Il est donc très facile d'utiliser cette valeur pour effectuer un traitement quelconque. Considérez par exemple le paramètre **x** de la fonction **updateLabel()**, dans notre exemple.

Le widget **Scale** constitue une interface très intuitive et attrayante pour proposer différents réglages aux utilisateurs de vos programmes. Nous allons à présent l'incorporer en plusieurs exemplaires dans une nouvelle classe de widget : un panneau de contrôle destiné à choisir la fréquence, la phase et l'amplitude pour un mouvement vibratoire, dont nous afficherons ensuite le graphique élongation/temps à l'aide du widget **oscilloGraphe** construit dans les pages précédentes.

13.3.2 Construction d'un panneau de contrôle à trois curseurs

Comme le précédent, le script que nous décrivons ci-dessous est destiné à être sauvegardé dans un module, que vous nommerez cette fois **curseurs.py**. Les classes que vous sauvegardez ainsi seront réutilisées (par importation) dans une application de synthèse que nous décrirons un peu plus loin[49]. Nous attirons votre attention sur le fait que le code ci-dessous peut être raccourci de différentes manières (Nous y reviendrons). Nous ne l'avons pas optimisé d'emblée, parce que cela nécessiterait d'y incorporer un concept supplémentaire (les expressions *lambda*), ce que nous préférons éviter pour l'instant.

Vous savez déjà que les lignes de code placées à la fin du script permettent de tester son fonctionnement. Vous devriez obtenir une fenêtre semblable à celle-ci :




```
from Tkinter import *
from math import pi

class ChoixVibra(Frame):
    """Curseurs pour choisir fréquence, phase & amplitude d'une vibration"""
    def __init__(self, boss =None, coul ='red'):
        Frame.__init__(self)        # constructeur de la classe parente
        # Initialisation de quelques attributs d'instance :
        self.freq, self.phase, self.ampl, self.coul = 0, 0, 0, coul
        # Variable d'état de la case à cocher :
        self.chk = IntVar()                # 'objet-variable' Tkinter
        Checkbutton(self, text='Afficher', variable=self.chk,
                    fg = self.coul, command = self.setCurve).pack(side=LEFT)
        # Définition des 3 widgets curseurs :
        Scale(self, length=150, orient=HORIZONTAL, sliderlength =25,
              label ='Fréquence (Hz) :', from_=1., to=9., tickinterval =2,
              resolution =0.25,
              showvalue =0, command = self.setFrequency).pack(side=LEFT)
        Scale(self, length=150, orient=HORIZONTAL, sliderlength =15,
              label ='Phase (degrés) :', from_=-180, to=180, tickinterval =90,
              showvalue =0, command = self.setPhase).pack(side=LEFT)
        Scale(self, length=150, orient=HORIZONTAL, sliderlength =25,
              label ='Amplitude :', from_=1, to=9, tickinterval =2,
              showvalue =0, command = self.setAmplitude).pack(side=LEFT)

    def setCurve(self):
        self.event_generate('<Control-Z>')

    def setFrequency(self, f):
        self.freq = float(f)
        self.event_generate('<Control-Z>')

    def setPhase(self, p):
        pp =float(p)
        self.phase = pp*2*pi/360        # conversion degrés -> radians
        self.event_generate('<Control-Z>')

    def setAmplitude(self, a):
        self.ampl = float(a)
        self.event_generate('<Control-Z>')

```

49 Vous pourriez bien évidemment aussi enregistrer plusieurs classes dans un même module.

```
42. #### Code pour tester la classe : ###
43.
44. if __name__ == '__main__':
45.     def afficherTout(event=None):
46.         lab.configure(text = '%s - %s - %s - %s' %
47.                          (fra.chk.get(), fra.freq, fra.phase, fra.ampl))
48.     root = Tk()
49.     fra = ChoixVibra(root,'navy')
50.     fra.pack(side =TOP)
51.     lab = Label(root, text ='test')
52.     lab.pack()
53.     root.bind('<Control-Z>', afficherTout)
54.     root.mainloop()
```

Ce panneau de contrôle permettra à vos utilisateurs de régler aisément la valeur des paramètres indiqués (fréquence, phase & amplitude), lesquels pourront alors servir à commander l'affichage de graphiques élongation/temps dans un *widget* de la classe **OscilloGraphe()** construite précédemment, comme nous le montrerons dans l'application de synthèse.

Commentaires :

- Ligne 6 : La méthode "constructeur" utilise un paramètre optionnel **coul**. Ce paramètre permettra de choisir une couleur pour le graphique soumis au contrôle du widget. Le paramètre **boss** sert à réceptionner la référence d'une fenêtre maîtresse éventuelle (voir plus loin).
- Ligne 7 : Activation du constructeur de la classe parente (pour hériter de sa fonctionnalité).
- Ligne 9 : Déclaration de quelques variables d'instance. Leurs vraies valeurs seront déterminées par les méthodes des lignes 29 à 40 (gestionnaires d'événements).
- Ligne 11 : Cette instruction instancie un objet de la classe **IntVar()**, laquelle fait partie du module *Tkinter* au même titre que les classes similaires **DoubleVar()**, **StringVar()** et **BooleanVar()**. Toutes ces classes permettent de définir des "*variables Tkinter*", lesquels sont en fait des objets, mais qui se se comportent comme des variables à l'intérieur des widgets *Tkinter*.
  Ainsi l'objet référencé dans **self.chk** contient l'équivalent d'une variable de type entier, dans un format utilisable par *Tkinter*. Pour accéder à sa valeur depuis Python, il faut utiliser des méthodes spécifiques de cette classe d'objets : la méthode **set()** permet de lui assigner une valeur, et la méthode **get()** permet de la récupérer (ce que l'on met en pratique à la ligne 47).
- Ligne 12 : L'option **variable** de l'objet **checkbutton** est associée à la *variable Tkinter* définie à la ligne précédente. (Nous ne pouvons pas référencer directement une variable ordinaire dans la définition d'un widget *Tkinter*, parce que *Tkinter* lui-même est écrit dans un langage qui n'utilise pas les mêmes conventions que Python pour formater ses variables. Les objets construits à partir des classes de *variables Tkinter* sont donc nécessaires pour assurer l'interface).
- Ligne 13 : L'option **command** désigne la méthode que le système doit invoquer lorsque l'utilisateur effectue un clic de souris dans la case à cocher.
- Lignes 14 à 24 : Ces lignes définissent les trois *widgets* curseurs, en trois instructions similaires. Il serait plus élégant de programmer tout ceci en une seule instruction, répétée trois fois à l'aide d'une boucle. Cela nécessiterait cependant de faire appel à un concept que nous n'avons pas encore expliqué (les fonctions/expressions *lamdba*), et la définition du gestionnaire d'événements associé à ces *widgets* deviendrait elle aussi plus complexe. Conservons donc pour cette fois des instructions séparées : nous nous efforcerons d'améliorer tout cela plus tard.

- Lignes 26 à 40 : Les 4 *widgets* définis dans les lignes précédentes possèdent chacun une option **command**. Pour chacun d'eux, la méthode invoquée dans cette option **command** est différente : la case à cocher active la méthode **setCurve()**, le premier curseur active la méthode **setFrequency()**, le second curseur active la méthode **setPhase()**, et le troisième curseur active la méthode **setAmplitude()**. Remarquez bien au passage que l'option **command** des *widgets* **Scale** transmet un argument à la méthode associée (la position actuelle du curseur), alors que la même option **command** ne transmet rien dans le cas du *widget* **Checkbutton**.

  Ces 4 méthodes (qui sont donc les gestionnaires des événements produits par la case à cocher et les trois curseurs) provoquent elles-mêmes chacune l'émission d'un nouvel événement[50], en faisant appel à la méthode **event_generate()**, elle-même associée au présent *widget*.

  Lorsque cette méthode est invoquée, Python envoie au système d'exploitation exactement le même message-événement que celui qui se produirait si l'utilisateur enfonçait simultanément les touches <Ctrl>, <Maj> et <Z> de son clavier.
  Nous produisons ainsi un message-événement bien particulier, qui peut être détecté et traité par un gestionnaire d'événement associé à un autre *widget* (voir page suivante). De cette manière, nous mettons en place ***un véritable système de communication entre widgets*** : chaque fois que l'utilisateur exerce une action sur notre panneau de contrôle, celui-ci génère un événement spécifique, qui signale cette action à l'attention des autres *widgets* présents.

  Note : nous aurions pu choisir une autre combinaison de touches (ou même carrément un autre type d'événement). Notre choix s'est porté sur celle-ci parce qu'il y a vraiment très peu de chances que l'utilisateur s'en serve alors qu'il examine notre programme. Nous pourrons cependant produire nous-mêmes un tel événement au clavier à titre de test, lorsque le moment sera venu de vérifier le gestionnaire de cet événement, que nous mettrons en place par ailleurs.

- Lignes 42 à 54 : Comme nous l'avions déjà fait pour **oscillo.py**, nous complétons ce nouveau module par quelques lignes de code au niveau principal. Ces lignes permettent de tester le bon fonctionnement de la classe : elles ne s'exécutent que si on lance le module directement, comme une application à part entière. Veillez à utiliser vous-même cette technique dans vos propres modules, car elle constitue une bonne pratique de programmation : l'utilisateur de modules construits ainsi peut en effet (re)découvrir très aisément leur fonctionnalité (en les exécutant) et la manière de s'en servir (en analysant ces quelques lignes de code).

  Dans ces lignes de test, nous construisons une fenêtre principale **root** qui contient deux widgets : un widget de la nouvelle classe **ChoixVibra()** et un widget de la classe **Label()**.

  A la ligne 53, nous associons à la fenêtre principale un gestionnaire d'événement : tout événement du type spécifié déclenche désormais un appel de la fonction **afficherTout()**.
  Cette fonction est donc notre gestionnaire d'événement spécialisé, qui est sollicité chaque fois qu'un événement de type <Maj-Ctrl-Z> est détecté par le système d'exploitation.

  Comme nous l'avons déjà expliqué plus haut, nous avons fait en sorte que de tels événements soient produits par les objets de la classe **ChoixVibra()**, chaque fois que l'utilisateur modifie l'état de l'un ou l'autre des trois curseurs, ou celui de la case à cocher.

50 En fait, on devrait plutôt appeler cela un message (qui est lui-même la notification d'un événement). Veuillez relire à ce sujet les explications de la page 83 : ***Programmes pilotés par des événements***.

Conçue seulement pour effectuer un test, la fonction **afficherTout()** ne fait rien d'autre que provoquer l'affichage des valeurs des variables associées à chacun de nos quatre widgets, en (re)configurant l'option **text** d'un widget de classe **Label()**.

- Ligne 47, expression **fra.chk.get()** : nous avons vu plus haut que la variable mémorisant l'état de la case à cocher est un *objet-variable Tkinter.* Python ne peut pas lire directement le contenu d'une telle variable, qui est en réalité un objet-interface. Pour en extraire la valeur, il faut donc faire usage d'une méthode spécifique de cette classe d'objets : la méthode **get()**.

Propagation des événements

Le mécanisme de communication décrit ci-dessus respecte la hiérarchie de classes des *widgets*. Vous aurez noté que la méthode qui déclenche l'événement est associée au *widget* dont nous sommes en train de définir la classe, par l'intermédiaire de **self**. En général, un message-événement est en effet associé à un *widget* particulier (par exemple, un clic de souris sur un bouton est associé à ce bouton), ce qui signifie que le système d'exploitation va d'abord examiner s'il existe un gestionnaire pour ce type d'événement, qui soit lui aussi associé à ce *widget*. S'il en existe un, c'est celui-là qui est activé, et la propagation du message s'arrête. Sinon, le message-événement est "présenté" successivement aux *widgets* maîtres, dans l'ordre hiérarchique, jusqu'à ce qu'un gestionnaire d'événement soit trouvé, ou bien jusqu'à ce que la fenêtre principale soit atteinte.

Les événements correspondant à des frappes sur le clavier (telle la combinaison de touches <Maj-Ctrl-Z> utilisée dans notre exercice) sont cependant toujours expédiés directement à la fenêtre principale de l'application. Dans notre exemple, le gestionnaire de cet événement doit donc être associé à la fenêtre **root**.

Exercices :

e 171. Votre nouveau widget hérite des propriétés de la classe **Frame()**. Vous pouvez donc modifier son aspect en modifiant les options par défaut de cette classe, à l'aide de la méthode **configure()**. Essayez par exemple de faire en sorte que le panneau de contrôle soit entouré d'une bordure de 4 pixels ayant l'aspect d'un sillon (*bd = 4, relief = GROOVE*). Si vous ne comprenez pas bien ce qu'il faut faire, inspirez-vous du script **oscillo.py** (ligne 10).

e 172. Si l'on assigne la valeur **1** à l'option **showvalue** des widgets **Scale()**, la position précise du curseur par rapport à l'échelle est affichée en permanence. Activez donc cette fonctionnalité pour le curseur qui contrôle le paramètre "phase".

e 173. L'option **troughcolor** des widgets **Scale()** permet de définir la couleur de leur glissière. Utilisez cette option pour faire en sorte que la couleur des glissières des 3 curseurs soit celle qui est utilisée comme paramètre lors de l'instanciation de votre nouveau widget.

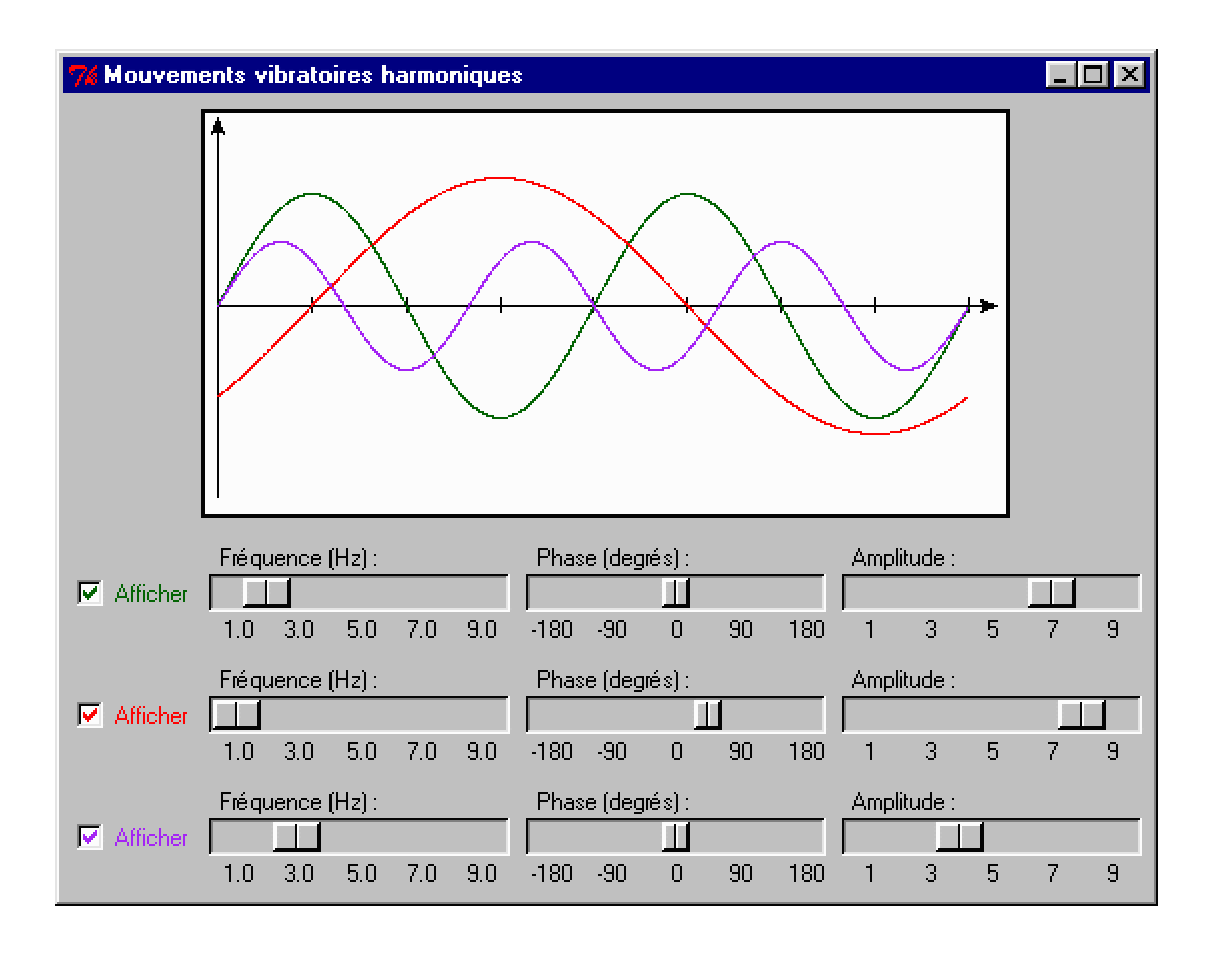
e 174. Modifiez le script de telle manière que les widgets curseurs soient écartés davantage les uns des autres (options **padx** et **pady** de la méthode **pack()**).

13.4 Intégration de widgets composites dans une application synthèse

Dans les exercices précédents, nous avons construit deux nouvelles classes de *widgets* : le *widget* **OscilloGraphe()**, canevas spécialisé pour le dessin de sinusoïdes, et le *widget* **ChoixVibra()**, panneau de contrôle à trois curseurs permettant de choisir les paramètres d'une vibration.

Ces *widgets* sont désormais disponibles dans les modules **oscillo.py** et **curseurs.py**[51]

Nous allons à présent les utiliser dans une application synthèse, qui pourrait illustrer votre cours de physique : un *widget* **OscilloGraphe()** y affiche un, deux, ou trois graphiques superposés, de couleurs différentes, chacun d'entre eux étant soumis au contrôle d'un *widget* **ChoixVibra()** :



Le script correspondant est reproduit ci-après.

Nous attirons votre attention sur la technique mise en oeuvre pour provoquer un rafraîchissement de l'affichage dans le canevas par l'intermédiaire d'un événement, chaque fois que l'utilisateur effectue une action quelconque au niveau de l'un des panneaux de contrôle.

Rappelez-vous que les applications destinées à fonctionner dans une interface graphique doivent être conçues comme des "programmes pilotés par les événements" (voir page 83).

51 Il va de soi que nous pourrions rassembler toutes les classes que nous construisons dans un seul module.

En préparant cet exemple, nous avons arbitrairement décidé que l'affichage des graphiques serait déclenché par un événement particulier, tout à fait similaire à ceux que génère le système d'exploitation lorsque l'utilisateur accomplit une action quelconque. Dans la gamme (très étendue) d'événements possibles, nous en avons choisi un qui ne risque guère d'être utilisé pour d'autres raisons, pendant que notre application fonctionne : la combinaison de touches <Maj-Ctrl-Z>.

Lorsque nous avons construit la classe de *widgets* **ChoixVibra()**, nous y avons donc incorporé les instructions nécessaires pour que de tels événements soient générés, chaque fois que l'utilisateur actionne l'un des curseurs ou modifie l'état de la case à cocher. Nous allons à présent définir le gestionnaire de cet événement et l'inclure dans notre nouvelle classe : nous l'appellerons **montreCourbes()** et il se chargera de rafraîchir l'affichage. Étant donné que l'événement concerné est du type <enfoncement d'une touche>, nous devrons cependant le détecter au niveau de la fenêtre principale de l'application.

```
from oscillo import *
from curseurs import *

class ShowVibra(Frame):
    """Démonstration de mouvements vibratoires harmoniques"""
    def __init__(self, boss =None):
        Frame.__init__(self)        # constructeur de la classe parente
        self.couleur = ['dark green', 'red', 'purple']
        self.trace = [0]*3          # liste des tracés (courbes à dessiner)
        self.controle = [0]*3       # liste des panneaux de contrôle

        # Instanciation du canevas avec axes X et Y :
        self.gra = OscilloGraphe(self, larg =400, haut=200)
        self.gra.configure(bg ='white', bd=2, relief=SOLID)
        self.gra.pack(side =TOP, pady=5)

        # Instanciation de 3 panneaux de contrôle (curseurs) :
        for i in range(3):
            self.controle[i] = ChoixVibra(self, self.couleur[i])
            self.controle[i].pack()

        # Désignation de l'événement qui déclenche l'affichage des tracés :
        self.master.bind('<Control-Z>', self.montreCourbes)
        self.master.title('Mouvements vibratoires harmoniques')
        self.pack()

    def montreCourbes(self, event):
        """(Ré)Affichage des trois graphiques élongation/temps"""
        for i in range(3):

            # D'abord, effacer le tracé précédent (éventuel) :
            self.gra.delete(self.trace[i])

            # Ensuite, dessiner le nouveau tracé :
            if self.controle[i].chk.get():
                self.trace[i] = self.gra.traceCourbe(
                                    coul = self.couleur[i],
                                    freq = self.controle[i].freq,
                                    phase = self.controle[i].phase,
                                    ampl = self.controle[i].ampl)

#### Code pour tester la classe : ###

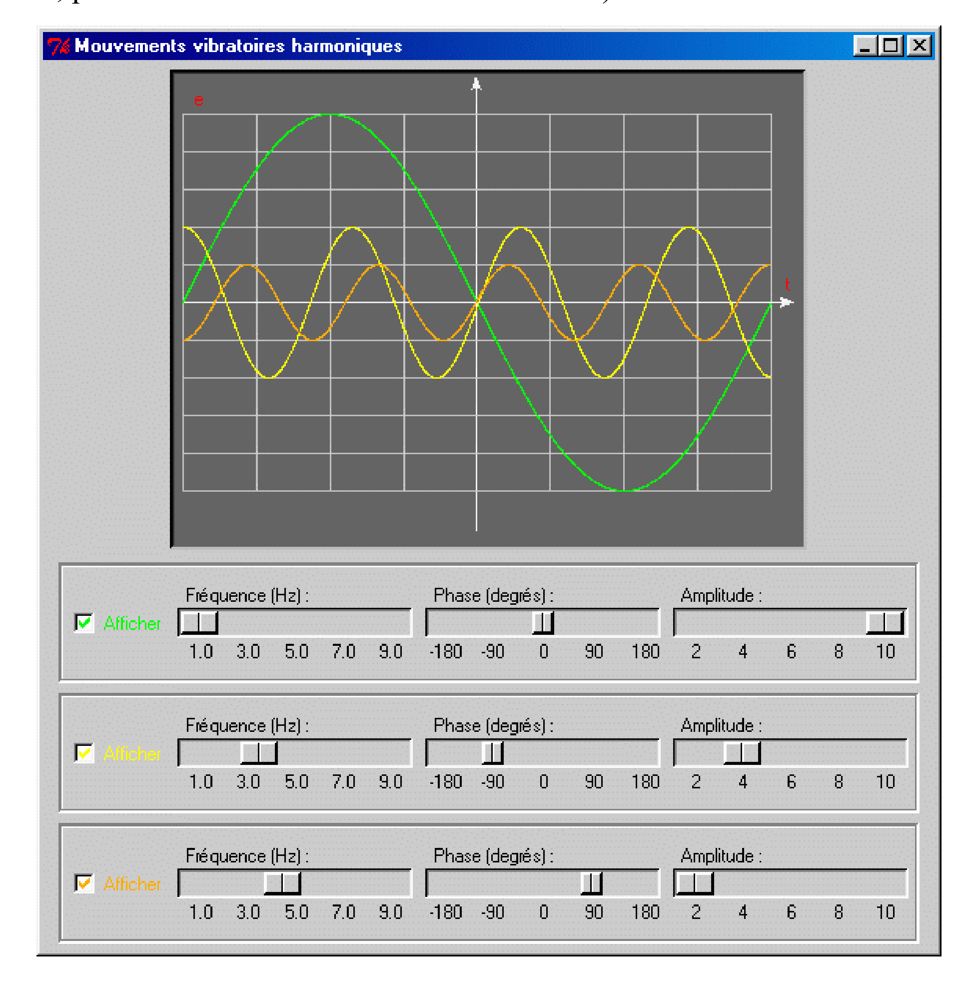
if __name__ == '__main__':
    ShowVibra().mainloop()
```

Commentaires :

- Lignes 1-2 : Nous pouvons nous passer d'importer le module *Tkinter* : chacun de ces deux modules s'en charge déjà.
- Ligne 4 : Puisque nous commençons à connaître les bonnes techniques, nous décidons de construire l'application elle-même sous la forme d'une classe, dérivée de la classe **Frame()** : ainsi nous pourrons plus tard l'intégrer toute entière dans d'autres projets, si le coeur nous en dit.
- Lignes 8-10 : Définition de quelques variables d'instance (3 listes) : les trois courbes tracées seront des objets graphiques, dont les couleurs sont pré-définies dans la liste **self.couleur** ; nous devons préparer également une liste **self.trace** pour mémoriser les références de ces objets graphiques, et enfin une liste **self.controle** pour mémoriser les références des trois panneaux de contrôle.
- Lignes 13 à 15 : Instanciation du widget d'affichage. Étant donné que la classe **OscilloGraphe()** a été obtenue par dérivation de la classe **Canvas()**, il est toujours possible de configurer ce widget en redéfinissant les options spécifiques de cette classe (ligne 13).
- Lignes 18 à 20 : Pour instancier les trois widgets "panneau de contrôle", on utilise une boucle. Leurs références sont mémorisées dans la liste **self.controle** préparée à la ligne 10. Ces panneaux de contrôle sont instanciés comme esclaves du présent widget, par l'intermédiaire du paramètre **self**. Un second paramètre leur transmet la couleur du tracé à contrôler.
- Lignes 23-24 : Au moment de son instanciation, chaque widget *Tkinter* reçoit automatiquement un attribut **master** qui contient la référence de la fenêtre principale de l'application. Cet attribut se révèle particulièrement utile si la fenêtre principale a été instanciée implicitement par Tkinter, comme c'est le cas ici.
  Rappelons en effet que lorsque nous démarrons une application en instanciant directement un widget tel que *Frame,* par exemple (c'est ce que nous avons fait à la ligne 4), *Tkinter* instancie automatiquement une fenêtre maîtresse pour ce widget (un objet de la classe **Tk()**).
  Comme cet objet a été créé automatiquement, nous ne disposons d'aucune référence dans notre code pour y accéder, si ce n'est par l'intermédiaire de l'attribut **master** que *Tkinter* associe automatiquement à chaque widget.
  Nous nous servons de cette référence pour redéfinir le bandeau-titre de la fenêtre principale (à la ligne 24), et pour y attacher un gestionnaire d'événement (à la ligne 23).
- Lignes 27 à 40 : La méthode décrite ici est le gestionnaire des événements <Maj-Ctrl-Z> spécifiquement déclenchés par nos widgets **ChoixVibra()** (ou "panneaux de contrôle"), chaque fois que l'utilisateur exerce une action sur un curseur ou une case à cocher. Dans tous les cas, les graphiques éventuellement présents sont d'abord effacés (ligne 28) à l'aide de la méthode **delete()** : le widget **OscilloGraphe()** a hérité cette méthode de sa classe parente **Canvas()**.
  Ensuite, de nouvelles courbes sont retracées, pour chacun des panneaux de contrôle dont on a coché la case "Afficher". Chacun des objets ainsi dessinés dans le canevas possède un numéro de référence, renvoyé par la méthode **traceCourbe()** de notre widget **OscilloGraphe()**.
  Les numéros de référence de nos dessins sont mémorisés dans la liste **self.trace**.
  Ils permettent d'effacer individuellement chacun d'entre eux (cfr. instruction de la ligne 28).
- Lignes 38-40 : Les valeurs de fréquence, phase & amplitude que l'on transmet à la méthode **traceCourbe()** sont les attributs d'instance correspondants de chacun des trois panneaux de contrôle, eux-mêmes mémorisés dans la liste **self.controle**. Nous pouvons récupérer ces attributs en utilisant la qualification des noms par points.

Exercices :

e 175. Modifiez le script, de manière à obtenir l'aspect ci-dessous (écran d'affichage avec grille de référence, panneaux de contrôle entourés d'un sillon) :



e 176. Modifiez le script, de manière à faire apparaître et contrôler 4 graphiques au lieu de trois. Pour la couleur du quatrième graphique, choisissez par exemple : 'blue', 'navy', 'maroon', ...

e 177. Aux lignes 33-35, nous récupérons les valeurs des fréquence, phase & amplitude choisies par l'utilisateur sur chacun des trois panneaux de contrôle, en accédant directement aux attributs d'instance correspondants. Python autorise ce raccourci - et c'est bien pratique – mais cette technique enfreint l'une des recommandations des théoriciens de la "programmation orientée objet", qui préconisent que l'accès aux propriétés des objets soit toujours pris en charge par des méthodes spécifiques. Pour respecter cette recommandation, ajoutez à la classe **ChoixVibra()** une méthode supplémentaire que vous appellerez **valeurs()**, et qui retournera un tuple contenant les valeurs de la fréquence, la phase et l'amplitude choisies. Les lignes 33 à 35 du présent script pourront alors être remplacées par quelque chose comme :

```
freq, phase, ampl = self.control[i].valeurs() n
```

e 178. Écrivez une petite application qui fait apparaître une fenêtre avec un canevas et un widget curseur (Scale). Dans le canevas, dessinez un cercle, dont l'utilisateur pourra faire varier la taille à l'aide du curseur.

Chapitre 14 : Et pour quelques widgets de plus ...

Les pages qui suivent contiennent des indications et et des exemples complémentaires qui pourront vous être utiles pour le développement de vos projets personnels. Il ne s'agit évidemment pas d'une documentation de référence complète sur *Tkinter*. Pour en savoir plus, vous devrez tôt ou tard consulter des ouvrages spécialisés, comme par exemple l'excellent *Python and Tkinter programming* de John E. Grayson, dont vous trouverez la référence complète à la page 8.

14.1 Les "boutons radio"

Les widgets "boutons radio" permettent de proposer à l'utilisateur un ensemble de choix mutuellement exclusifs. On les appelle ainsi par analogie avec les boutons de sélection que l'on trouvait jadis sur les postes de radio. Ces boutons étaient conçus de telle manière qu'un seul à la fois pouvait être enfoncé : tous les autres ressortaient automatiquement.

La caractéristique essentielle de ces widgets est qu'on les utilise toujours par groupes. Tous les boutons radio faisant partie d'un même groupe sont associés à une seule et même *variable Tkinter*, mais chacun d'entre eux se voit aussi attribuer une *valeur* particulière.
Lorsque l'utilisateur sélectionne l'un des boutons, la valeur correspondant à ce bouton est affectée à la *variable Tkinter* commune.




```
from Tkinter import *

class RadioDemo(Frame):
    """Démo : utilisation de widgets 'boutons radio'"""
    def __init__(self, boss =None):
        """Création d'un champ d'entrée avec 4 boutons radio"""
        Frame.__init__(self)
        self.pack()
        # Champ d'entrée contenant un petit texte :
        self.texte = Entry(self, width =30, font ="Arial 14")
        self.texte.insert(END, "La programmation, c'est génial")
        self.texte.pack(padx =8, pady =8)
        # Nom français et nom technique des quatre styles de police :
        stylePoliceFr =["Normal", "Gras", "Italique", "Gras/Italique"]
        stylePoliceTk =["normal", "bold", "italic"  , "bold italic"]
        # Le style actuel est mémorisé dans un 'objet-variable' Tkinter ;
        self.choixPolice = StringVar()
        self.choixPolice.set(stylePoliceTk[0])
        # Création des quatre 'boutons radio' :
        for n in range(4):
            bout = Radiobutton(self,
                               text = stylePoliceFr[n],
                               variable = self.choixPolice,
                               value = stylePoliceTk[n],
                               command = self.changePolice)
            bout.pack(side =LEFT, padx =5)

    def changePolice(self):
        """Remplacement du style de la police actuelle"""
        police = "Arial 15 " + self.choixPolice.get()
        self.texte.configure(font =police)

if __name__ == '__main__':
    RadioDemo().mainloop()
```

Commentaires :

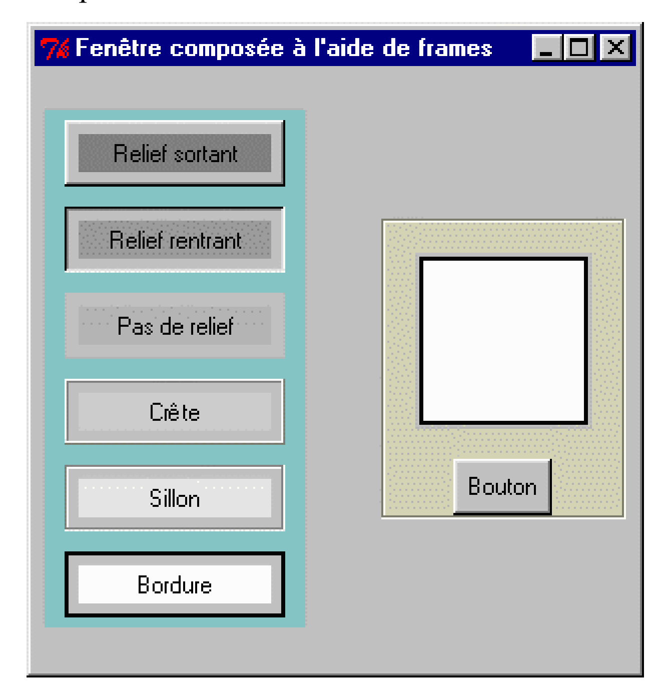
- Ligne 3 : Cette fois encore, nous préférons construire notre petite application comme une classe dérivée de la classe **Frame()**, ce qui nous permettrait éventuellement de l'intégrer sans difficulté dans une application plus importante.
- Ligne 8 : En général, on applique les méthodes de positionnement des widgets (**pack()**, **grid()**, ou **place()**) après instanciation de ceux-ci, ce qui permet de choisir librement leur disposition à l'intérieur des fenêtres maîtresses. Comme nous le montrons ici, il est cependant tout à fait possible de déjà prévoir ce positionnement dans le constructeur du widget.
- Ligne 11 : Les widgets de la classe **Entry** disposent de plusieurs méthodes pour accéder à la chaîne de caractères affichée. La méthode **get()** permet de récupérer la chaîne entière. La méthode **delete()** permet d'en effacer tout ou partie (cfr. projet "Code des couleurs", page 166). La méthode **insert()** permet d'insérer de nouveaux caractères à un emplacement quelconque (c.à.d. au début, à la fin, ou même à l'intérieur d'une chaîne préexistante éventuelle). Cette méthode s'utilise donc avec deux arguments, le premier indiquant l'emplacement de l'insertion (utilisez 0 pour insérer au début, END pour insérer à la fin, ou encore un indice numérique).
- Lignes 14-15 : Plutôt que de les instancier dans des instructions séparées, nous préférons créer nos quatre boutons à l'aide d'une boucle. Les options spécifiques à chacun d'eux sont d'abord préparées dans les deux listes **stylePoliceFr** et **stylePoliceTk** : la première contient les petits textes qui devront s'afficher en regard de chaque bouton, et la seconde les valeurs qui devront leur être associées.
- Lignes 17-18 : Comme expliqué à la page précédente, les quatre boutons forment un groupe autour d'une variable commune. Cette variable prendra la valeur associée au bouton radio que l'utilisateur décidera de choisir. Nous ne pouvons cependant pas utiliser une variable ordinaire pour remplir ce rôle, parce que les attributs internes des objets *Tkinter* ne sont accessibles qu'au travers de méthodes spécifiques. Une fois de plus, nous utilisons donc ici un objet-variable *Tkinter,* de type 'chaîne de caractères', que nous instancions à partir de la classe **StringVar()**., et auquel nous donnons une valeur par défaut à la ligne 18.
- Lignes 20 à 26 : Instanciation des quatre boutons radio. Chacun d'entre eux se voit attribuer une étiquette et une valeur différentes, mais tous sont associés à la même *variable Tkinter* commune (**self.choixPolice**). Tous invoquent également la même méthode **self.changePolice()**, chaque fois que l'utilisateur effectue un clic de souris sur l'un ou l'autre.
- Lignes 28 à 31 : Le changement de police s'obtient par re-configuration de l'option **font** du widget **Entry**. Cette option attend un tuple contenant le nom de la police, sa taille, et éventuellement son style. Si le nom de la police ne contient pas d'espaces, le tuple peut aussi être remplacé par une chaîne de caractères. Exemples :

  ```
  ('Arial', 12, 'italic')
  ('Helvetica', 10)
  ('Times New Roman', 12, 'bold italic')
  "Verdana 14 bold"
  "President 18 italic"
  ```

  Voir également les exemples de la page 205.

14.2 Utilisation des cadres (frames) pour la composition d'une fenêtre

Vous avez déjà abondamment utilisé la classe de widgets **Frame()** ("cadre", en français), notamment pour créer de nouveaux widgets complexes par dérivation.



Le petit script ci-dessous vous montre l'utilité de cette même classe pour regrouper des ensembles de widgets et les disposer d'une manière déterminée dans une fenêtre. Il vous démontre également l'utilisation de certaines options décoratives (bordures, relief, etc.).

Pour composer la fenêtre ci-contre, nous avons utilisé deux cadres **f1** et **f2**, de manière à réaliser deux groupes de widgets bien distincts, l'un à gauche et l'autre à droite. Nous avons coloré ces deux cadres pour bien les mettre en évidence, mais ce n'est évidemment pas indispensable.

Le cadre **f1** contient lui-même 6 autres cadres, qui contiennent chacun un widget de la classe **Label()**. Le cadre **f2** contient un widget **Canvas()** et un widget **Button()**. Les couleurs et garnitures sont de simples options.

```
from Tkinter import *

fen = Tk()
fen.title("Fenêtre composée à l'aide de frames")
fen.geometry("300x300")

f1 = Frame(fen, bg = '#80c0c0')
f1.pack(side =LEFT, padx =5)

fint = [0]*6
for (n, col, rel, txt) in [(0, 'grey50', RAISED, 'Relief sortant'),
                           (1, 'grey60', SUNKEN, 'Relief rentrant'),
                           (2, 'grey70', FLAT, 'Pas de relief'),
                           (3, 'grey80', RIDGE, 'Crête'),
                           (4, 'grey90', GROOVE, 'Sillon'),
                           (5, 'grey100', SOLID, 'Bordure')]:
    fint[n] = Frame(f1, bd =2, relief =rel)
    e = Label(fint[n], text =txt, width =15, bg =col)
    e.pack(side =LEFT, padx =5, pady =5)
    fint[n].pack(side =TOP, padx =10, pady =5)

f2 = Frame(fen, bg ='#d0d0b0', bd =2, relief =GROOVE)
f2.pack(side =RIGHT, padx =5)

can = Canvas(f2, width =80, height =80, bg ='white', bd =2, relief =SOLID)
can.pack(padx =15, pady =15)
bou =Button(f2, text='Bouton')
bou.pack()

fen.mainloop()
```

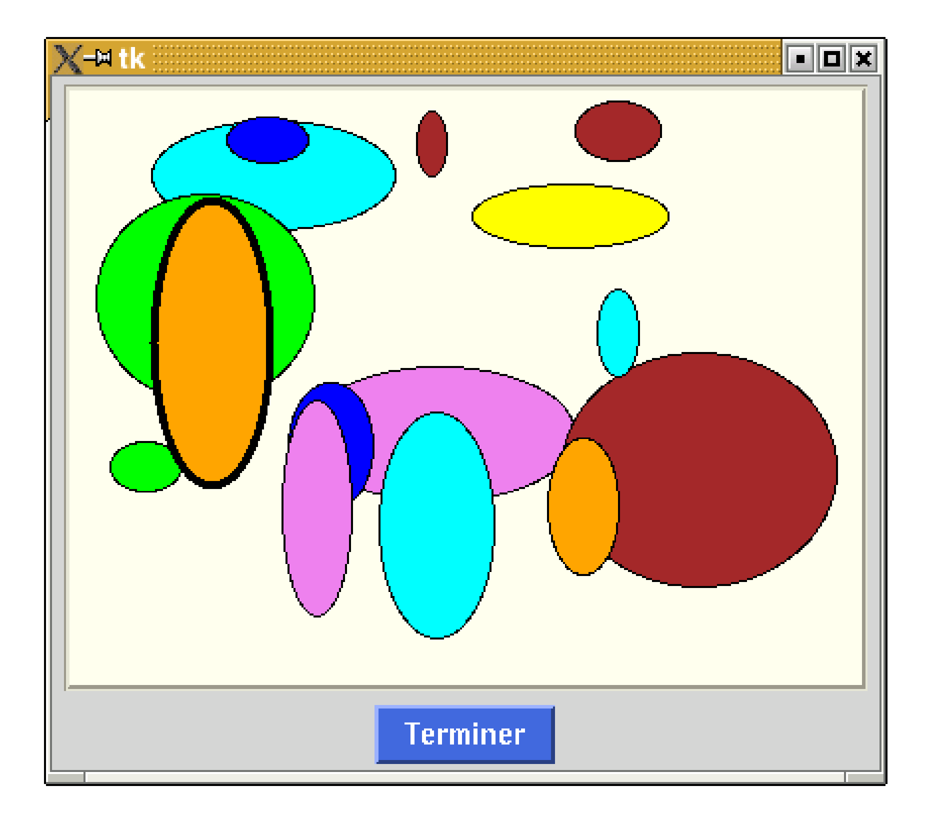
- Lignes 3 à 5 : Afin de simplifier au maximum la démonstration, nous ne programmons pas cet exemple comme une nouvelle classe. Remarquez à la ligne 5 l'utilité de la méthode **geometry()** pour fixer les dimensions de la fenêtre principale.
- Ligne 7 : Instanciation du cadre de gauche. La couleur de fond (une variété de bleu cyan) est déterminée par l'argument **bg** (*background*). Cette chaîne de caractères contient en notation hexadécimale la description des trois composantes rouge, verte et bleue de la teinte que l'on souhaite obtenir : Après le caractère # signalant que ce qui suit est une valeur numérique hexadécimale, on trouve trois groupes de deux symboles alphanumériques. Chacun de ces groupes représente un nombre compris entre 1 et 255. Ainsi **80** correspond à 128, et **c0** correspond à 192 en notation décimale. Dans notre exemple, les composantes rouge, verte et bleue de la teinte à représenter valent donc respectivement 128, 192 & 192.
  En application de cette technique descriptive, le noir serait obtenu avec **#000000**, le blanc avec **#ffffff**, le rouge pur avec **#ff0000**, un bleu sombre avec **#000050**, etc.
- Ligne 8 : Puisque nous lui appliquons la méthode **pack()**, le cadre sera automatiquement dimensionné par son contenu. L'option **side =LEFT** le positionnera à gauche dans sa fenêtre maîtresse. L'option **padx =5** ménagera un espace de 5 pixels à sa gauche et à sa droite (nous pouvons traduire "padx" par "espacement horizontal").
- Ligne 10 : Dans le cadre **f1** que nous venons de préparer, nous avons l'intention de regrouper 6 autres cadres similaires contenant chacun une étiquette. Le code correspondant sera plus simple et plus efficient si nous instancions ces widgets dans une liste plutôt que dans des variables indépendantes. Nous préparons donc cette liste avec 6 éléments que nous remplacerons plus loin.
- Lignes 11 à 16 : Pour construire nos 6 cadres similaires, nous allons parcourir une liste de 6 tuples contenant les caractéristiques particulières de chaque cadre. Chacun de ces tuples est constitué de 4 éléments : un indice, une constante *Tkinter* définissant un type de relief, et deux chaînes de caractères décrivant respectivement la couleur et le texte de l'étiquette.
  La boucle **for** effectue 6 itérations pour parcourir les 6 éléments de la liste. A chaque itération, le contenu d'un des tuples est affecté aux variables **n**, **col, rel** et **txt** (et ensuite les instructions des lignes 17 à 20 sont exécutées). Le parcours d'une liste de tuples à l'aide d'une boucle **for** constitue donc une construction particulièrement compacte, qui permet de réaliser de nombreuses affectations avec un très petit nombre d'instructions.
- Ligne 17 : Les 6 cadres sont instanciés comme des éléments de la liste **fint**. Chacun d'entre eux est agrémenté d'une bordure décorative de 2 pixels de large, avec un certain effet de relief.
- Lignes 18-20 : Les étiquettes ont toutes la même taille, mais leurs textes et leurs couleurs de fond diffèrent. Du fait de l'utilisation de la méthode **pack()**, c'est la dimension des étiquettes qui détermine la taille des petits cadres. Ceux-ci à leur tour déterminent la taille du cadre qui les regroupe (le cadre **f1**). Les options **padx** et **pady** permettent de réserver un petit espace autour de chaque étiquette, et un autre autour de chaque petit cadre. L'option **side =TOP** positionne les 6 petits cadres les uns en dessous des autres dans le cadre conteneur **f1**.
- Lignes 22-23 : Préparation du cadre **f2** (cadre de droite). Sa couleur sera une variété de jaune, et nous l'entourerons d'une bordure décorative ayant l'aspect d'un sillon.
- Lignes 25 à 28 : Le cadre **f2** contiendra un canevas et un bouton. Notez encore une fois l'utilisation des options **padx** et **pady** pour ménager des espaces autour des widgets (Considérez par exemple le cas du bouton, pour lequel cette option n'a pas été utilisée : de ce fait, il entre en contact avec la bordure du cadre qui l'entoure). Comme nous l'avons fait pour les cadres, nous avons placé une bordure autour du canevas. Sachez que d'autres widgets acceptent également ce genre de décoration : boutons, champs d'entrée, etc.

14.3 Comment déplacer des dessins à l'aide de la souris

Le widget canevas est l'un des points forts de la bibliothèque graphique *Tkinter*. Il intègre en effet un grand nombre de dispositifs très efficaces pour manipuler des dessins. Le script ci-après est destiné à vous montrer quelques techniques de base. Si vous voulez en savoir plus, notamment en ce qui concerne la manipulation de dessins composés de plusieurs parties, veuillez consulter l'un ou l'autre ouvrage de référence traitant de *Tkinter*.

Au démarrage de notre petite application, une série de dessins sont tracés au hasard dans un canevas (il s'agit en l'occurrence de simples ellipses colorées). Vous pouvez déplacer n'importe lequel de ces dessins en les "saisissant" à l'aide de votre souris.

Lorsqu'un dessin est déplacé, il passe à l'avant-plan par rapport aux autres, et sa bordure apparaît plus épaisse pendant toute la durée de sa manipulation.



Pour bien comprendre la technique utilisée, vous devez vous rappeler qu'un logiciel utilisant une interface graphique est un logiciel "piloté par les événements" (revoyez au besoin les explications de la page 83). Dans cette application, nous allons mettre en place un mécanisme qui réagit aux événements : "enfoncement du bouton gauche de la souris", "déplacement de la souris, le bouton gauche restant enfoncé", "relâchement du bouton gauche".

Ces événements sont générés par le système d'exploitation et pris en charge par l'interface *Tkinter*. Notre travail de programmation consistera donc simplement à les associer à des gestionnaires différents (fonctions ou méthodes).

```
# Exemple montrant comment faire en sorte que les objets dessinés dans un
# canevas puissent être manipulés à l'aide de la souris

from Tkinter import *
from random import randrange

class Draw(Frame):
    "classe définissant la fenêtre principale du programme"
    def __init__(self):
        Frame.__init__(self)
     # mise en place du canevas - dessin de 15 ellipses colorés :
        self.c = Canvas(self, width =400, height =300, bg ='ivory')
        self.c.pack(padx =5, pady =3)
        for i in range(15):
            # tirage d'une couleur au hasard :
            coul =['brown','red','orange','yellow','green','cyan','blue',
                   'violet', 'purple'][randrange(9)]
            # tracé d'une ellipse avec coordonnées aléatoires :
            x1, y1 = randrange(300), randrange(200)
            x2, y2 = x1 + randrange(10, 150), y1 + randrange(10, 150)
            self.c.create_oval(x1, y1, x2, y2, fill =coul)
        # liaison d'événements <souris> au widget <canevas> :
        self.c.bind("<Button-1>", self.mouseDown)
        self.c.bind("<Button1-Motion>", self.mouseMove)
        self.c.bind("<Button1-ButtonRelease>", self.mouseUp)
        # mise en place d'un bouton de sortie :
        b_fin = Button(self, text ='Terminer', bg ='royal blue', fg ='white',
                       font =('Helvetica', 10, 'bold'), command =self.quit)
        b_fin.pack(pady =2)
        self.pack()

    def mouseDown(self, event):
        "Op. à effectuer quand le bouton gauche de la souris est enfoncé"
        self.currObject =None
        # event.x et event.y contiennent les coordonnées du clic effectué :
        self.x1, self.y1 = event.x, event.y
        # <find_closest> renvoie la référence du dessin le plus proche :
        self.selObject = self.c.find_closest(self.x1, self.y1)
        # modification de l'épaisseur du contour du dessin :
        self.c.itemconfig(self.selObject, width =3)
        # <lift> fait passer le dessin à l'avant-plan :
        self.c.lift(self.selObject)

    def mouseMove(self, event):
        "Op. à effectuer quand la souris se déplace, bouton gauche enfoncé"
        x2, y2 = event.x, event.y
        dx, dy = x2 -self.x1, y2 -self.y1
        if self.selObject:
            self.c.move(self.selObject, dx, dy)
            self.x1, self.y1 = x2, y2

    def mouseUp(self, event):
        "Op. à effectuer quand le bouton gauche de la souris est relâché"
        if self.selObject:
            self.c.itemconfig(self.selObject, width =1)
            self.selObject =None

if __name__ == '__main__':
    Draw().mainloop()
```

Commentaires :

Le script contient essentiellement la définition d'une classe graphique dérivée de **Frame()**.

Comme c'est souvent le cas pour les programmes exploitant les classes d'objets, le corps principal du script se résume à une seule instruction composée, dans laquelle on réalise deux opérations consécutives : instanciation d'un objet de la classe définie précédemment, et activation de sa méthode **mainloop()** (laquelle démarre l'observateur d'événements).

Le constructeur de la classe **Draw()** présente une structure qui doit vous être devenue familière, à savoir : appel au constructeur de la classe parente, puis mise en place de divers widgets.

Dans le widget canevas, nous instancions 15 dessins sans nous préoccuper de conserver leurs références dans des variables. Nous pouvons procéder ainsi parce que *Tkinter* conserve lui-même une référence interne pour chacun de ces objets. (Si vous travaillez avec d'autres bibliothèques graphiques, vous devrez probablement prévoir une mémorisation de ces références).

Les dessins sont de simples ellipses colorées. Leur couleur est choisie au hasard dans une liste de 9 possibilités, l'indice de la couleur choisie étant déterminé par la fonction **randrange()** importée du module **random**.

Le mécanisme d'interaction est installé ensuite : on associe les trois identificateurs d'événements **<Button-1>**, **<Button1-Motion>** et **<Button1-ButtonRelease>** concernant le widget canevas, aux noms des trois méthodes choisies comme gestionnaires d'événements.

Lorsque l'utilisateur enfonce le bouton gauche de sa souris, la méthode **mouseDown()** est donc activée, et le système d'exploitation lui transmet en argument un objet **event**, dont les attributs **x** et **y** contiennent les coordonnées du curseur souris dans le canevas, déterminées au moment du clic.

Nous mémorisons directement ces coordonnées dans les variables d'instance **self.x1** et **self.x2**, car nous en aurons besoin par ailleurs. Ensuite, nous utilisons la méthode **find_closest()** du widget canevas, qui nous renvoie la référence du dessin le plus proche. (Note : cette méthode bien pratique renvoie toujours une référence, même si le clic de souris n'a pas été effectué à l'intérieur du dessin).

Le reste est facile : la référence du dessin sélectionné est mémorisée dans une variable d'instance, et nous pouvons faire appel à d'autres méthodes du widget canevas pour modifier ses caractéristiques. En l'occurrence, nous utilisons les méthodes **itemconfig()** et **lift()** pour épaissir son contour et le faire passer à l'avant-plan.

Le "transport" du dessin est assuré par la méthode **mouseMove()**, invoquée à chaque fois que la souris se déplace alors que son bouton gauche est resté enfoncé. L'objet **event** contient cette fois encore les coordonnées du curseur souris, au terme de ce déplacement. Nous nous en servons pour calculer les différences entre ces nouvelles coordonnées et les précédentes, afin de pouvoir les transmettre à la méthode **move()** du widget canevas, qui effectuera le transport proprement dit.

Nous ne pouvons cependant faire appel à cette méthode que s'il existe effectivement un objet sélectionné, et il nous faut veiller également à mémoriser les nouvelles coordonnées acquises.

La méthode **mouseUp()** termine le travail. Lorsque le dessin transporté est arrivé à destination, il reste à annuler la sélection et rendre au contour son épaisseur initiale. Ceci ne peut être envisagé que s'il existe effectivement une sélection, bien entendu.

14.4 Python Mega Widgets

Les modules ***Pmw*** constituent une extension intéressante de *Tkinter*. Entièrement écrits en Python, ils contiennent toute une bibliothèque de widgets composites, construits à partir des classes de base de *Tkinter*. Dotés de fonctionnalités très étendues, ces widgets peuvent se révéler fort précieux pour le développement rapide d'applications complexes. Si vous souhaitez les utiliser, sachez cependant que les modules *Pmw* ne font pas partie de l'installation standard de Python : vous devrez donc toujours vérifier leur présence sur les machines cibles de vos programmes.

Il existe un grand nombre de ces méga-widgets. Nous n'en présenterons ici que quelques-uns parmi les plus utiles. Vous pouvez rapidement vous faire une idée plus complète de leurs multiples possibilités, en essayant les scripts de démonstration qui les accompagnent (lancez par exemple le script **all.py** , situé dans le répertoire .../Pmw/demos).

14.4.1 "Combo Box"

Les méga-widgets s'utilisent aisément. La petite application ci-après vous montre comment mettre en oeuvre un widget de type **ComboBox** (boîte de liste combinée à un champ d'entrée). Nous l'avons configuré de la manière la plus habituelle (avec une boîte de liste déroulante).



Lorsque l'utilisateur de notre petit programme choisit une couleur dans la liste déroulante (il peut aussi entrer un nom de couleur directement dans le champ d'entrée), cette couleur devient automatiquement la couleur de fond pour la fenêtre maîtresse.

Dans cette fenêtre maîtresse, nous avons ajouté un libellé et un bouton, afin de vous montrer comment vous pouvez accéder à la sélection opérée précédemment dans le *ComboBox* lui-même (le bouton provoque l'affichage du nom de la dernière couleur choisie).

```
from Tkinter import *
import Pmw

def changeCoul(col):
    fen.configure(background = col)

def changeLabel():
    lab.configure(text = combo.get())

couleurs = ('navy', 'royal blue', 'steelblue1', 'cadet blue',
            'lawn green', 'forest green', 'dark red',
            'grey80','grey60', 'grey40', 'grey20')

fen = Pmw.initialise()
bou = Button(fen, text ="Test", command =changeLabel)
bou.grid(row =1, column =0, padx =8, pady =6)
lab = Label(fen, text ='néant', bg ='ivory')
lab.grid(row =1, column =1, padx =8)

combo = Pmw.ComboBox(fen, labelpos = NW,
                     label_text = 'Choisissez la couleur :',
                     scrolledlist_items = couleurs,
                     listheight = 150,
                     selectioncommand = changeCoul)
combo.grid(row =2, columnspan =2, padx =10, pady =10)

fen.mainloop()
```

Commentaires :

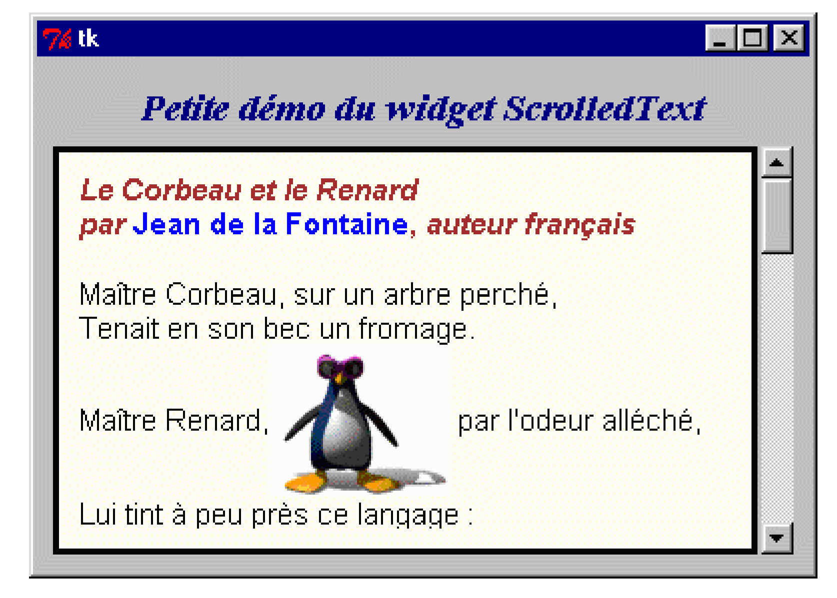
- Lignes 1 & 2 : On commence par importer les composants habituels de *Tkinter*, ainsi que le module **Pmw**.
- Ligne 14 : Pour créer la fenêtre maîtresse, il faut utiliser de préférence la méthode **Pmw.initialise()**, plutôt que d'instancier directement un objet de la classe **Tk()**. Cette méthode veille en effet à mettre en place tout ce qui est nécessaire afin que les widgets esclaves de cette fenêtre puissent être détruits correctement lorsque la fenêtre elle-même sera détruite. Cette méthode installe également un meilleur gestionnaire des messages d'erreurs.
- Ligne 12 : L'option **labelpos** détermine l'emplacement du libellé qui accompagne le champ d'entrée. Dans notre exemple, nous l'avons placé au-dessus, mais vous pourriez préférer le placer ailleurs, à gauche par exemple (**labelpos = W**). Notez que cette option est indispensable si vous souhaitez un libellé (pas de valeur par défaut).
- Ligne 14 : L'option **selectioncommand** transmet un argument à la fonction invoquée : l'item sélectionné dans la boîte de liste. Vous pourrez également retrouver cette sélection à l'aide de la méthode **get()**, comme nous le faisons à la ligne 8 pour actualiser le libellé.

14.4.2 "Scrolled Text"

Ce méga-widget étend les possibilités du widget **Text** sandard, en lui associant un cadre, un libellé (titre) et des barres de défilement.

Comme le démontrera le petit script ci-dessous, il sert fondamentalement à afficher des textes, mais ceux-ci peuvent être mis en forme et intégrer des images.

Vous pouvez également rendre "cliquables" les éléments affichés (textes ou images), et vous en servir pour déclencher toutes sortes de mécanismes.



Dans l'application qui génère la figure ci-dessus, par exemple, le fait de cliquer sur le nom "Jean de la Fontaine" provoque le défilement automatique du texte (*scrolling*), jusqu'à ce qu'une rubrique décrivant cet auteur devienne visible dans le widget (Voir page suivante le script correspondant).

D'autres fonctionnalités sont présentes, mais nous ne présenterons ici que les plus fondamentales. Veuillez donc consulter les démos et exemples accompagnant *Pmw* pour en savoir davantage.

Gestion du texte affiché : Vous pouvez accéder à n'importe quelle portion du texte pris en charge par le widget grâce à deux concepts complémentaires, les ***index*** et les ***balises*** :

- Chaque caractère du texte affiché est référencé par un ***index***, lequel doit être une chaîne contenant deux valeurs numériques reliées par un point (ex : "5.2"). Ces deux valeurs indiquent respectivement le numéro de ligne et le numéro de colonne où se situe le caractère.
- N'importe quelle portion du texte peut être associée à une ou plusieurs ***balise(s)***, dont vous choisissez librement le nom et les propriétés. Celles-ci vous permettent de définir la police, les couleurs d'avant- et d'arrière-plan, les événements associés, etc.

Note : Pour la bonne compréhension du script ci-dessous, veuillez considérer que le texte de la fable traitée doit être accessible, dans un fichier nommé "CorbRenard.txt".

```
from Tkinter import *
import Pmw

def action(event=None):
    """défilement du texte jusqu'à la balise <cible>"""
    index = st.tag_nextrange('cible', '0.0', END)
    st.see(index[0])

# Instanciation d'une fenêtre contenant un widget ScrolledText :
fen = Pmw.initialise()
st = Pmw.ScrolledText(fen,
                      labelpos =N,
                      label_text ="Petite démo du widget ScrolledText",
                      label_font ='Times 14 bold italic',
                      label_fg = 'navy', label_pady =5,
                      text_font='Helvetica 11 normal', text_bg ='ivory',
                      text_padx =10, text_pady =10, text_wrap ='none',
                      borderframe =1,
                      borderframe_borderwidth =3,
                      borderframe_relief =SOLID,
                      usehullsize =1,
                      hull_width =370, hull_height =240)
st.pack(expand =YES, fill =BOTH, padx =8, pady =8)

# Définition de balises, liaison d'un gestionnaire d'événement au clic de souris :
st.tag_configure('titre', foreground ='brown', font ='Helvetica 11 bold italic')
st.tag_configure('lien', foreground ='blue', font ='Helvetica 11 bold')
st.tag_configure('cible', foreground ='forest green', font ='Times 11 bold')
st.tag_bind('lien', '<Button-1>', action)

titre ="""Le Corbeau et le Renard
par Jean de la Fontaine, auteur français
\n"""
auteur ="""
Jean de la Fontaine
écrivain français (1621-1695)
célèbre pour ses Contes en vers,
et surtout ses Fables, publiées
de 1668 à 1694."""

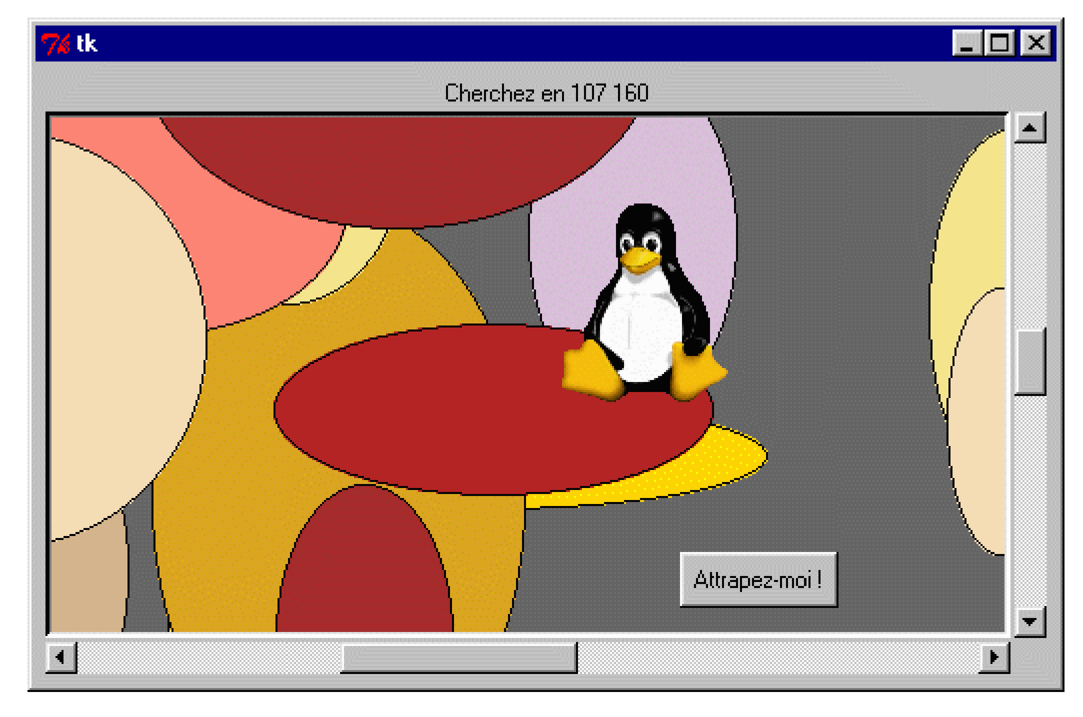
# Remplissage du widget Text (2 techniques) :
st.importfile('CorbRenard.txt')
st.insert('0.0', titre, 'titre')
st.insert(END, auteur, 'cible')
# Insertion d'une image :
photo =PhotoImage(file= 'Penguin.gif')
st.image_create('6.14', image =photo)
# Mise en oeuvre dynamique d'une balise :
st.tag_add('lien', '2.4', '2.23')

fen.mainloop()
```

- Lignes 4-7 : Cette fonction est un gestionnaire d'événement, qui est appelé lorsque l'utilisateur effectue un clic de souris sur le nom de l'auteur (cfr. lignes 27 & 29). A la ligne 6, on utilise la méthode **tag_nextrange()** du widget pour trouver les ***index*** de la portion de texte associée à la ***balise*** "cible". La recherche de ces index est limitée au domaine défini par les 2e & 3e arguments (dans notre exemple, on recherche du début à la fin du texte entier). La méthode **tag_nextrange()** renvoie une liste de deux index (ceux des premier et dernier caractères de la portion de texte associée à la balise "cible"). A la ligne 7, nous nous servons d'un seul de ces index (le premier) pour activer la méthode **see().** Celle-ci provoque un défilement automatique du texte (*scrolling*), de telle manière que le caractère correspondant à l'index transmis devienne visible dans le widget (avec en général un certain nombre des caractères qui suivent).
- Lignes 9 à 23 : Construction classique d'une fenêtre destinée à afficher un seul widget. Dans le code d'instanciation du widget, nous avons inclus un certain nombre d'options destinées à vous montrer une petite partie des nombreuses possibilités de configuration.
- Ligne 12 : L'option **labelpos** détermine l'emplacement du libellé (titre) par rapport à la fenêtre de texte. Les valeurs acceptées s'inspirent des lettres utilisées pour désigner les points cardinaux (N, S, E, W, ou encore NE, NW, SE, SW). Si vous ne souhaitez pas afficher un libellé, il vous suffit tout simplement de ne pas utiliser cette option.
- Lignes 13 à 15 : Le libellé n'est rien d'autre qu'un widget **Label** standard, intégré dans le widget composite **ScrolledText**. On peut accéder à toutes ses options de configuration, en utilisant la syntaxe qui est présentée dans ces lignes : on y voit qu'il suffit d'associer le préfixe **label_** au nom de l'option que l'on souhaite activer, pour définir aisément les couleurs d'avant- et d'arrière-plans, la police, la taille, et même l'espacement à réserver autour du widget (option **pady**).
- Lignes 16-17 : En utilisant une technique similaire à celle qui est décrite ci-dessus pour le libellé, on peut accéder aux options de configuration du widget **Text** intégré dans **ScrolledText.** Il suffit cette fois d'associer aux noms d'option le préfixe **text_**.
- Lignes 18 à 20 : Il est prévu un cadre (un widget **Frame**) autour du widget **Text**. L'option **borderframe = 1** permet de le faire apparaître. On accède ensuite à ses options de configuration d'une manière similaire à celle qui a été décrite ci-dessus pour **label_** et **text_**.
- Lignes 21-22 : Ces options permettent de fixer globalement les dimensions du widget. Une autre possibilité serait de définir plutôt les dimensions de son composant **Text** (par exemple à l'aide d'options telles que **text_width** et **text_height**), mais alors les dimensions globales du widget risqueraient de changer en fonction du contenu (apparition/disparition automatique de barres de défilement). *Note : le mot **hull** désigne le contenant global, c.à.d. le méga-widget lui-même.*
- Ligne 23 : Les options **expand = YES** et **fill = BOTH** de la méthode **pack()** indiquent que le widget concerné pourra être redimensionné à volonté, dans ses 2 dimensions horiz. et verticale.
- Lignes 26 à 29 : Ces lignes définissent les trois ***balises*** "titre", "lien" et "cible" ainsi que le formatage du texte qui leur sera associé. La ligne 29 précise en outre que le texte associé à la balise "lien" sera "cliquable", avec indication du gestionnaire d'événement correspondant.
- Ligne 42 : Importation de texte à partir d'un fichier. Note : Il est possible de préciser l'endroit exact où devra se faire l'insertion, en fournissant un index comme second argument.
- Lignes 43-44 : Ces instructions insèrent des fragments de texte (respectivement au début et à la fin du texte préexistant), en associant une balise à chacun d'eux.
- Ligne 49 : L'association des balises au texte est dynamique. A tout moment, vous pouvez activer une nouvelle association (comme nous le faisons ici en rattachant la balise "lien" à une portion de texte préexistante). *Note : pour "détacher" une balise, utilisez la méthode **tag_delete()**.*

14.4.3 "Scrolled Canvas"

Le script ci-après vous montre comment vous pouvez exploiter le méga-widget **ScrolledCanvas**, lequel étend les possibilités du widget **Canvas** standard en lui associant des barres de défilement, un libellé et un cadre. Notre exemple constitue en fait un petit jeu d'adresse, dans lequel l'utilisateur doit réussir à cliquer sur un bouton qui s'esquive sans cesse. (*Note : si vous éprouvez vraiment des difficultés pour l'attraper, commencez d'abord par dilater la fenêtre*).



Le widget **Canvas** est très versatile : il vous permet de combiner à volonté des dessins, des images *bitmap*, des fragments de texte, et même d'autres widgets, dans un espace parfaitement extensible. Si vous souhaitez développer l'un ou l'autre jeu graphique, c'est évidemment le widget qu'il vous faut apprendre à maîtriser en priorité.

Comprenez bien cependant que les indications que nous vous fournissons à ce sujet dans les présentes notes sont forcément très incomplètes. Leur objectif est seulement de vous aider à comprendre quelques concepts de base, afin que vous puissiez ensuite consulter les ouvrages de référence spécialisés dans de bonnes conditions.

Notre petite application se présente comme une nouvelle classe **FenPrinc()**, obtenue par dérivation à partir de la classe de méga-widgets **Pmw.ScrolledCanvas()**. Elle contient donc un grand canevas muni de barres de défilement, dans lequel nous commençons par planter un décor constitué de 80 ellipses de couleur dont l'emplacement et les dimensions sont tirés au hasard.

Nous y ajoutons également un petit clin d'oeil sous la forme d'une image bitmap, destinée avant tout à vous rappeler comment vous pouvez gérer ce type de ressource.

Nous y installons enfin un véritable widget : un simple bouton, en l'occurrence, mais la technique mise en oeuvre pourrait s'appliquer à n'importe quel autre type de widget, y compris un gros widget composite comme ceux que nous avons développés précédemment. Cette grande souplesse dans le développement d'applications complexes est l'un des principaux bénéfices apportés par le mode de programmation "orientée objet".

Le bouton s'anime dès qu'on l'a enfoncé une première fois. Dans votre analyse du script ci-après, soyez attentifs aux méthodes utilisées pour modifier les propriétés d'un objet existant.

```
from Tkinter import *
import Pmw
from random import randrange

Pmw.initialise()
coul =['sienna','maroon','brown','pink','tan','wheat','gold','orange','plum',
       'red','khaki','indian red','thistle','firebrick','salmon','coral']

class FenPrinc(Pmw.ScrolledCanvas):
    """Fenêtre principale : canevas extensible avec barres de défilement"""
    def __init__(self):
        Pmw.ScrolledCanvas.__init__(self,
                 usehullsize =1, hull_width =500, hull_height =300,
                 canvas_bg ='grey40', canvasmargin =10,
                 labelpos =N, label_text ='Attrapez le bouton !',
                 borderframe =1,
                 borderframe_borderwidth =3)
        # Les options ci-dessous doivent être précisées après initialisation :
        self.configure(vscrollmode ='dynamic', hscrollmode ='dynamic')
        self.pack(padx =5, pady =5, expand =YES, fill =BOTH)

        self.can = self.interior()        # accès au composant canevas
        # Décor : tracé d'une série d'ellipses aléatoires :
        for r in range(80):
            x1, y1 = randrange(-800,800), randrange(-800,800)
            x2, y2 = x1 + randrange(40,300), y1 + randrange(40,300)
            couleur = coul[randrange(0,16)]
            self.can.create_oval(x1, y1, x2, y2, fill=couleur, outline='black')
        # Ajout d'une petite image GIF :
        self.img = PhotoImage(file ='linux2.gif')
        self.can.create_image(50, 20, image =self.img)
        # Dessin du bouton à attraper :
        self.x, self.y = 50, 100
        self.bou = Button(self.can, text ="Start", command =self.start)
        self.fb = self.can.create_window(self.x, self.y, window =self.bou)
        self.resizescrollregion()

    def anim(self):
        if self.run ==0:
            return
        self.x += randrange(-60, 61)
        self.y += randrange(-60, 61)
        self.can.coords(self.fb, self.x, self.y)
        self.configure(label_text = 'Cherchez en %s %s' % (self.x, self.y))
        self.resizescrollregion()
        self.after(250, self.anim)

    def stop(self):
        self.run =0
        self.bou.configure(text ="Restart", command =self.start)

    def start(self):
        self.bou.configure(text ="Attrapez-moi !", command =self.stop)
        self.run =1
        self.anim()

##### Main Program ##############

if __name__ == '__main__':
    FenPrinc().mainloop()
```

Commentaires :

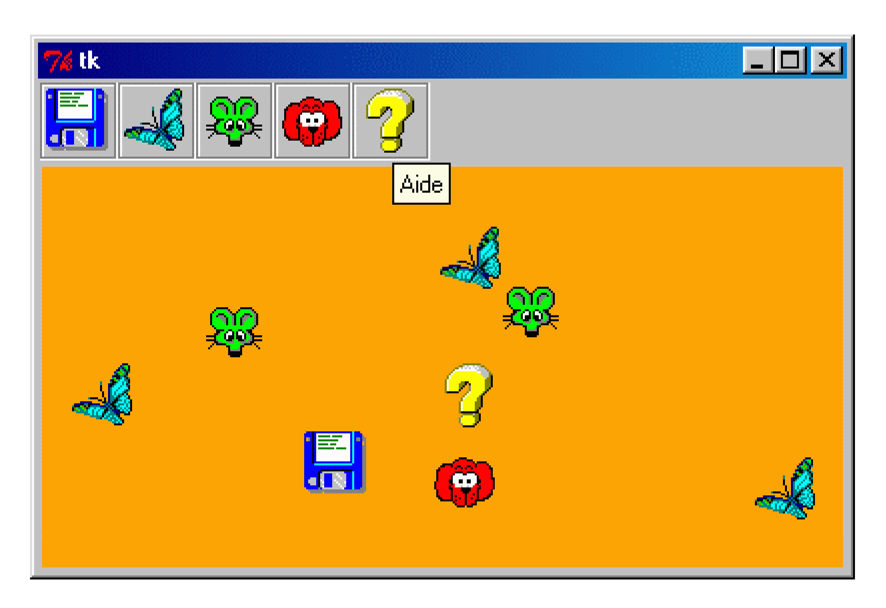
- Ligne 6 : Tous ces noms de couleurs sont acceptés par *Tkinter.* Vous pourriez bien évidemment les remplacer par des descriptions hexadécimales, comme nous l'avons expliqué page 187.
- Lignes 12 à 17 : Ces options sont très similaires à celles que nous avons décrites plus haut pour le widget ***ScrolledText***. Le présent méga-widget intègre un composant **Frame**, un composant **Label**, un composant **Canvas** et deux composants **Scrollbar**. On accède aux options de configuration de ces composants à l'aide d'une syntaxe qui relie le nom du composant et celui de l'option par l'intermédiaire d'un caractère "souligné".
- Ligne 19 : Ces options définissent le mode d'apparition des barres de défilement. En mode "*static*", elles sont toujours présentes. En mode "*dynamic*", elles disparaissent si les dimensions du canevas deviennent inférieures à celles de la fenêtre de visualisation.
- Ligne 22 : La méthode **interior()** renvoie la référence du composant **Canvas** intégré dans le méga-widget **ScrolledCanvas**. Les instructions suivantes (lignes 23 à 35) installent ensuite toute une série d'éléments dans ce canevas : des dessins, une image et un bouton.
- Lignes 25 à 27 : La fonction **randrange()** permet de tirer au hasard un nombre entier compris dans un certain intervalle (Veuillez vous référer aux explications de la page 138).
- Ligne 35 : C'est la méthode **create_window()** du widget **Canvas** qui permet d'y insérer n'importe quel autre widget (y compris un widget composite). Le widget à insérer doit cependant avoir été défini lui-même au préalable comme un esclave du canevas ou de sa fenêtre maîtresse.
  La méthode **create_window()** attend trois arguments : les coordonnées X et Y du point où l'on souhaite insérer le widget, et la référence de ce widget.
- Ligne 36 : La méthode **resizescrollregion()** réajuste la situation des barres de défilement de manière à ce qu'elles soient en accord avec la portion du canevas actuellement affichée.
- Lignes 38 à 46 : Cette méthode est utilisée pour l'animation du bouton. Après avoir repositionné le bouton au hasard à une certaine distance de sa position précédente, elle se ré-appelle elle-même après une pause de 250 millisecondes. Ce bouclage s'effectue sans cesse, aussi longtemps que la variable self.run contient une valeur non-nulle.
- Lignes 48 à 55 : Ces deux gestionnaires d'événement sont associés au bouton en alternance. Ils servent évidemment à démarrer et à arrêter l'animation.

14.4.4 Barres d'outils avec bulles d'aide - expressions lambda

De nombreux programmes comportent une ou plusieurs "barres d'outils" (*toolbar*) constituées de petits boutons sur lesquels sont représentés des pictogrammes (icônes). Cette façon de faire permet de proposer à l'utilisateur un grand nombre de commandes spécialisées, sans que celles-ci n'occupent une place excessive à l'écran (un petit dessin vaut mieux qu'un long discours, dit-on).

La signification de ces pictogrammes n'est cependant pas toujours évidente, surtout pour les utilisateurs néophytes. Il est donc vivement conseillé de compléter les barres d'outils à l'aide d'un système de bulles d'aide (*tool tips*), qui sont des petits messages explicatifs apparaissant automatiquement lorsque la souris survole les boutons concernés.

L'application décrite ci-après comporte une barre d'outils et un canevas. Lorsque l'utilisateur clique sur l'un des boutons de la barre, le pictogramme qu'il porte est recopié dans le canevas, à un emplacement choisi au hasard :



Dans notre exemple, chaque bouton apparaît entouré d'un sillon. Vous pouvez aisément obtenir d'autres aspects en choisissant judicieusement les options **relief** et **bd** (bordure) dans l'instruction d'instanciation des boutons. En particulier, vous pouvez choisir **relief = FLAT** et **bd =0** pour obtenir des petits boutons "plats", sans aucun relief.

La mise en place des bulles d'aide est un jeu d'enfant. Il suffit d'instancier un seul objet **Pmw.Balloon** pour l'ensemble de l'application, puis d'associer un texte à chacun des widgets auxquels on souhaite associer une bulle d'aide, en faisant appel autant de fois que nécessaire à la méthode **bind()** de cet objet.

Vous savez qu'en règle générale, on associe à chaque bouton une méthode ou une fonction particulière, laquelle se charge d'effectuer le travail lorsque le bouton est activé. Or dans l'application présente, tous les boutons doivent faire à peu près la même chose (recopier un dessin dans le canevas), la seule différence entre eux étant le dessin concerné. Pour simplifier notre code, nous voudrions donc pouvoir associer l'option **command** de tous nos boutons avec une seule et même méthode (la méthode **action()**), mais en lui transmettant à chaque fois la référence du bouton particulier utilisé, de manière à ce que l'action accomplie puisse être différente pour chaque.

Une difficulté se présente, parce que l'option **command** du *widget* **Button** accepte seulement une ***valeur*** ou une ***expression***, et non une ***instruction***. Il est donc permis de lui indiquer la référence d'une fonction, mais pas de l'invoquer véritablement en lui transmettant des arguments éventuels (c'est la raison pour laquelle on indique le nom de la fonction sans adjoindre de parenthèses).

On peut résoudre cette difficulté en faisant appel à une expression **lambda.** Ce mot réservé Python désigne une ***expression*** qui renvoie un objet fonction, similaire à ceux que vous créez avec l'instruction **def**, mais avec la différence que lambda étant une expression et non une instruction, on peut l'utiliser comme interface afin d'invoquer une fonction (avec passage d'arguments) là où ce n'est normalement pas possible.

Par exemple, `lambda ar1=b, ar2=c : bidule(ar1,ar2)` renvoie la référence d'une fonction anonyme qui aura elle-même invoqué la fonction **bidule()** en lui transmettant les arguments **b**et **c**.

Nous ne détaillerons pas davantage cette question ici, car elle déborde du cadre que nous nous sommes fixés pour ces notes. Si vous souhaitez en savoir plus, veuillez donc consulter l'un ou l'autre des ouvrages de référence cités dans la bibliographie.

```
from Tkinter import *
import Pmw
from random import randrange

# noms des fichiers contenant les icônes (format GIF):
images =('floppy_2','papi2','pion_1','pion_2','help_4')
textes =('sauvegarde','papillon','joueur 1','joueur 2','Aide')

class Application(Frame):
    def __init__(self):
        Frame.__init__(self)
        # Création d'un objet <bulle d'aide> (un seul suffit) :
        tip = Pmw.Balloon(self)
        # Création de la barre d'outils (c'est un simple cadre) :
        toolbar = Frame(self, bd =1)
        toolbar.pack(expand =YES, fill =X)
        # Nombre de boutons à construire :
        nBou = len(images)
        # Les icônes des boutons doivent être placées dans des variables
        # persistantes. Une liste fera l'affaire :
        self.photoI =[None]*nBou

        for b in range(nBou):
            # Création de l'icône (objet PhotoImage Tkinter) :
            self.photoI[b] =PhotoImage(file = images[b] +'.gif')
            # Création du bouton. On utilise une expression "lambda" pour
            # transmettre un argument à la méthode invoquée comme commande :
            bou = Button(toolbar, image =self.photoI[b], relief =GROOVE,
                         command = lambda arg =b: self.action(arg))
            bou.pack(side =LEFT)

            # association du bouton avec un texte d'aide (bulle) :
            tip.bind(bou, textes[b])

        self.ca = Canvas(self, width =400, height =200, bg ='orange')
        self.ca.pack()
        self.pack()

    def action(self, b):
        "l'icône du bouton b est recopiée dans le canevas"
        x, y = randrange(25,375), randrange(25,175)
        self.ca.create_image(x, y, image =self.photoI[b])

Application().mainloop()
```

14.5 Fenêtres avec menus

Nous allons décrire à présent la construction d'une fenêtre d'application dotée de différents types de menus "déroulants", chacun de ces menus pouvant être "détaché" de l'application principale pour devenir lui-même une petite fenêtre indépendante, comme dans l'illustration ci-dessous.

Cet exercice un peu plus long nous servira également de révision, et nous le réaliserons par étapes, en appliquant une stratégie de programmation que l'on appelle ***développement incrémental***.

Comme nous l'avons déjà expliqué précédemment[52], cette méthode consiste à commencer l'écriture d'un programme par une ébauche, qui ne comporte que quelques lignes seulement mais qui est déjà fonctionnelle. On teste alors cette ébauche soigneusement afin d'en éliminer les *bugs* éventuels. Lorsque l'ébauche fonctionne correctement, on y ajoute une fonctionnalité supplémentaire. On teste ce complément jusqu'à ce qu'il donne entière satisfaction, puis on en ajoute un autre, et ainsi de suite...



Cela ne signifie pas que vous pouvez commencer directement à programmer sans avoir au préalable effectué une analyse sérieuse du projet, dont au moins les grandes lignes devront être convenablement décrites dans un ***cahier des charges*** clairement rédigé.

Il reste également impératif de ***commenter*** convenablement le code produit, ***au fur et à mesure de son élaboration***. S'efforcer de rédiger de bons commentaires est en effet nécessaire, non seulement pour que votre code soit facile à lire (et donc à maintenir plus tard, par d'autres ou par vous-même), mais aussi pour que vous soyez forcés d'exprimer ce que vous souhaitez vraiment que la machine fasse (Cfr. *Erreurs sémantiques*, page 15)

Cahier des charges de l'exercice :

Notre application comportera simplement une barre de menus et un canevas. Les différentes rubriques et options des menus ne serviront qu'à faire apparaître des fragments de texte dans le canevas ou à modifier des détails de décoration, mais ce seront avant tout des exemples variés, destinés à donner un aperçu des nombreuses possibilités offertes par ce type de widget, accessoire indispensable de toute application moderne d'une certaine importance.

Nous souhaitons également que le code produit dans cet exercice soit bien structuré. Pour ce faire, nous ferons usage de deux classes : une classe pour l'application principale, et une autre pour la barre de menus. Nous voulons procéder ainsi afin de bien mettre en évidence la construction d'une application type incorporant plusieurs classes d'objets interactifs.

52 Voir page 16 : Recherche des erreurs et expérimentation

14.5.1 Première ébauche du programme :

Lorsque l'on construit l'ébauche d'un programme, il faut tâcher d'y faire apparaître le plus tôt possible la structure d'ensemble, avec les relations entre les principaux blocs qui constitueront l'application définitive. C'est ce que nous nous sommes efforcés de faire dans l'exemple ci-dessous :

```
from Tkinter import *

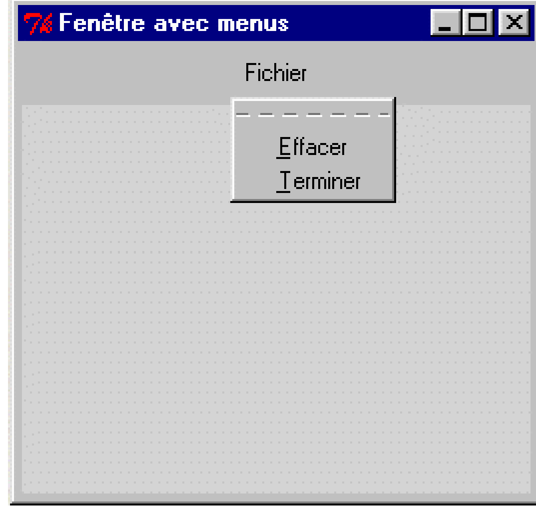
class MenuBar(Frame):
    """Barre de menus déroulants"""
    def __init__(self, boss =None):
        Frame.__init__(self, borderwidth =2)

        ##### Menu <Fichier> #####
        fileMenu = Menubutton(self, text ='Fichier')
        fileMenu.pack(side =LEFT)
        # Partie "déroulante" :
        me1 = Menu(fileMenu)
        me1.add_command(label ='Effacer', underline =0,
                        command = boss.effacer)
        me1.add_command(label ='Terminer', underline =0,
                        command = boss.quit)
        # Intégration du menu :
        fileMenu.configure(menu = me1)

class Application(Frame):
    """Application principale"""
    def __init__(self, boss =None):
        Frame.__init__(self)
        self.master.title('Fenêtre avec menus')
        mBar = MenuBar(self)
        mBar.pack()
        self.can = Canvas(self, bg='light grey', height=190,
                          width=250, borderwidth =2)
        self.can.pack()
        self.pack()

    def effacer(self):
        self.can.delete(ALL)

if __name__ == '__main__':
    app = Application()
    app.mainloop()
```



Veuillez donc encoder ces lignes et en tester l'exécution. Vous devriez obtenir une fenêtre avec un canevas gris clair surmonté d'une barre de menus. A ce stade, la barre de menus ne comporte encore que la seule rubrique "Fichier".

Cliquez sur la rubrique "fichier" pour faire apparaître le menu correspondant : l'option "Effacer" n'est pas encore fonctionnelle (elle servira à effacer le contenu du canevas), mais l'option "Terminer" devrait déjà vous permettre de fermer proprement l'application.

Comme tous les menus gérés par *Tkinter*, le menu que vous avez créé peut être converti en menu "flottant" : il suffit de cliquer sur la ligne pointillée apparaissant en-tête de menu. Vous obtenez ainsi une petite fenêtre satellite, que vous pouvez alors positionner où bon vous semble sur le bureau.

Analyse du script :

La structure de ce petit programme devrait désormais vous apparaître familière : afin que les classes définies dans ce script puissent éventuellement être (ré)utilisées dans d'autres projets par importation, comme nous l'avons déjà expliqué précédemment[53], le corps principal du programme (lignes 35 à 37) comporte l'instruction classique : `if __name__ == '__main__':`

Les deux instructions qui suivent consistent seulement à instancier un objet **app** et à faire fonctionner sa méthode **mainloop()**. Comme vous le savez certainement, nous aurions pu également condenser ces deux instructions en une seule.

L'essentiel du du programme se trouve cependant dans les définitions de classes qui précèdent :

La classe **MenuBar()** contient la description de la barre de menus. Dans l'état présent du script, elle se résume à une ébauche de constructeur.

- Ligne 5 : Le paramètre **boss** réceptionne la référence de la fenêtre maîtresse du widget au moment de son instanciation. Cette référence va nous permettre d'invoquer les méthodes associées à cette fenêtre maîtresse, aux lignes 14 & 16.
- Ligne 6 : Activation obligatoire du constructeur de la classe parente.
- Ligne 9 : Instanciation d'un widget de la classe **Menubutton()**, défini comme un "esclave" de **self** (c.à.d. l'objet composite "barre de menus" dont nous sommes occupés à définir la classe). Comme l'indique son nom, ce type de widget se comporte un peu comme un bouton : une action se produit lorsque l'on clique dessus.
- Ligne 12 : Afin que cette action consiste en l'apparition véritable d'un menu, il reste encore à définir celui-ci : ce sera encore un nouveau widget, de la classe **Menu()** cette fois, défini lui-même comme un "esclave" du widget *Menubutton* instancié à la ligne 9.
- Lignes 13 à 16 : On peut appliquer aux widgets de la classe **Menu()** un certain nombre de méthodes spécifiques, chacune d'elles acceptant de nombreuses options. Nous utilisons ici la méthode **add_command()** pour installer dans le menu les deux ***items*** "Effacer" et "Terminer". Nous y intégrons tout de suite l'option **underline**, qui sert à définir un ***raccourci clavier*** : cette option indique en effet lequel des caractères de l'item doit apparaître souligné à l'écran. L'utilisateur sait alors qu'il lui suffit de frapper ce caractère au clavier pour que l'action correspondant à cet item soit activée (comme s'il avait cliqué dessus à l'aide de la souris).
L'action à déclencher lorsque l'utilisateur sélectionne l'item est désignée par l'option **command**. Dans notre script, les commandes invoquées sont toutes les deux ***des méthodes de la fenêtre maîtresse***, dont la référence aura été transmise au présent widget au moment de son instanciation par l'intermédiaire du paramètre **boss**. La méthode **effacer()**, que nous définissons nous-même plus loin, servira à vider le canevas. La méthode prédéfinie **quit()** provoque la sortie de la boucle **mainloop()** et donc l'arrêt du réceptionnaire d'événements associé à la fenêtre d'application.
- Ligne 18 : Lorsque les items du menu ont été définis, il reste encore à reconfigurer le widget maître *Menubutton* de manière à ce que son option "menu" désigne effectivement le *Menu* que nous venons de construire. En effet, nous ne pouvions pas déjà préciser cette option lors de la définition initiale du widget *Menubutton*, puisqu'à ce stade le *Menu* n'existait pas encore. Nous ne pouvions pas non plus définir le widget *Menu* en premier lieu, puisque celui-ci doit être défini comme un "esclave" du widget *Menubutton*. Il faut donc bien procéder en trois étapes comme nous l'avons fait, en faisant appel à la méthode **configure()**. (Cette méthode peut être appliquée à n'importe quel widget préexistant pour en modifier l'une ou l'autre option).

53 Voir page 164: Modules contenant des bibliothèques de classes

La classe **Application()** contient la description de la fenêtre principale du programme ainsi que les méthodes gestionnaires d'événements qui lui sont associées.

- Ligne 20 : Nous préférons faire dériver notre application de la classe **Frame()**, qui présente de nombreuses options, plutôt que de la classe primordiale **Tk()**. De cette manière, l'application toute entière est encapsulée dans un widget, lequel pourra éventuellement être intégré par la suite dans une application plus importante. Rappelons que de toute manière, *Tkinter* instanciera automatiquement une fenêtre maîtresse de type **Tk()** pour contenir de cette *Frame*.
- Lignes 23-24 : Après l'indispensable activation du constructeur de la classe parente, nous utilisons l'attribut **master** que *Tkinter* associe automatiquement à chaque widget, pour référencer la fenêtre principale de l'application (la fenêtre maîtresse dont nous venons de parler au paragraphe précédent) et en redéfinir le bandeau-titre.
- Lignes 25 à 29 : Instanciation de deux widgets esclaves pour notre *Frame* principale.
  La "barre de menus" est évidemment le widget défini dans l'autre classe.
- Ligne 30 : Comme n'importe quel autre widget, notre *Frame* principale doit être mise en place.
- Lignes 32-33 : La méthode servant à effacer le canevas est définie dans la classe présente (puisque l'objet canevas en fait partie), mais elle est invoquée par l'option *command* d'un widget esclave défini dans l'autre classe. Comme nous l'avons expliqué plus haut, ce widget esclave reçoit la référence de son widget maître par l'intermédiaire du paramètre **boss**.
  Toutes ces références sont hiérarchisées à l'aide de la qualification des noms par points.

14.5.2 Ajout de la rubrique "Musiciens"

Continuez le développement de ce petit programme, en ajoutant les lignes suivantes dans le constructeur de la classe **MenuBar()** (après la ligne 18) :

```
##### Menu <Musiciens> #####
self.musi = Menubutton(self, text ='Musiciens')
self.musi.pack(side =LEFT, padx ='3')
# Partie "déroulante" du menu <Musiciens> :
me1 = Menu(self.musi)
me1.add_command(label ='17e siècle', underline =1,
                foreground ='red', background ='yellow',
                font =('Comic Sans MS', 11),
                command = boss.showMusi17)
me1.add_command(label ='18e siècle', underline =1,
                foreground='royal blue', background ='white',
                font =('Comic Sans MS', 11, 'bold'),
                command = boss.showMusi18)
# Intégration du menu :
self.musi.configure(menu = me1)
```

... ainsi que les définitions de méthodes suivantes à la classe **Application()** (après la ligne 33) :

```
def showMusi17(self):
    self.can.create_text(10, 10, anchor =NW, text ='H. Purcell',
                font=('Times', 20, 'bold'), fill ='yellow')

def showMusi18(self):
    self.can.create_text(245, 40, anchor =NE, text ="W. A. Mozart",
                font =('Times', 20, 'italic'), fill ='dark green')
```

Lorsque vous y aurez ajouté toutes ces lignes, sauvegardez le script et exécutez-le.



Votre barre de menus comporte à présent une rubrique supplémentaire : la rubrique "Musiciens".

Le menu correspondant propose deux items qui sont affichés avec des couleurs et des polices personnalisées. Vous pourrez vous inspirer de ces techniques décoratives pour vos projets personnels. A utiliser avec modération !

Les commandes que nous avons associées à ces items sont évidemment simplifiées afin de ne pas alourdir l'exercice : elles provoquent l'affichage de petits textes sur le canevas.

Analyse du script

Les seules nouveautés introduites dans ces lignes concernent l'utilisation de polices de caractères bien déterminées (option **font**), ainsi que de couleurs pour l'avant-plan (option **foreground**) et le fond (option **background**) des textes affichés.

Veuillez noter encore une fois l'utilisation de l'option **underline** pour désigner les caractères correspondant à des raccourcis claviers (en n'oubliant pas que la numérotation des caractères d'une chaîne commence à partir de zéro), et surtout que l'option **command** de ces widgets accède aux méthodes de l'autre classe, par l'intermédiaire de la référence mémorisée dans l'attribut **boss**.

La méthode **create_text()** du canevas doit être utilisée avec deux arguments numériques, qui sont les coordonnées X et Y d'un point dans le canevas. Le texte transmis sera positionné par rapport à ce point, en fonction de la valeur choisie pour l'option **anchor** : Celle-ci détermine comment le fragment de texte doit être "ancré" au point choisi dans le canevas, par son centre, par son coin supérieur gauche, etc., en fonction d'une syntaxe qui utilise l'analogie des points cardinaux géographiques (NW = angle supérieur gauche, SE = angle inférieur droit, CENTER = centre, etc.)

14.5.3 Ajout de la rubrique "Peintres" :

Cette nouvelle rubrique est construite d'une manière assez semblable à la précédente, mais nous lui avons ajouté une fonctionnalité supplémentaire : des menus "en cascade". Veuillez donc ajouter les lignes suivantes dans le constructeur de la classe **MenuBar()** :

```
##### Menu <Peintres> #####
self.pein = Menubutton(self, text ='Peintres')
self.pein.pack(side =LEFT, padx='3')
# Partie "déroulante" :
me1 = Menu(self.pein)
me1.add_command(label ='classiques', state=DISABLED)
me1.add_command(label ='romantiques', underline =0,
                command = boss.showRomanti)
# Sous-menu pour les peintres impressionistes :
me2 = Menu(me1)
me2.add_command(label ='Claude Monet', underline =7,
                command = boss.tabMonet)
me2.add_command(label ='Auguste Renoir', underline =8,
                command = boss.tabRenoir)
me2.add_command(label ='Edgar Degas', underline =6,
                command = boss.tabDegas)
# Intégration du sous-menu :
me1.add_cascade(label ='impressionistes', underline=0, menu =me2)
# Intégration du menu :
self.pein.configure(menu =me1)
```

... et les définitions suivantes dans la classe **Application()** :

```
def showRomanti(self):
    self.can.create_text(245, 70, anchor =NE, text = "E. Delacroix",
                font =('Times', 20, 'bold italic'), fill ='blue')

def tabMonet(self):
    self.can.create_text(10, 100, anchor =NW, text = 'Nymphéas à Giverny',
                font =('Technical', 20), fill ='red')

def tabRenoir(self):
    self.can.create_text(10, 130, anchor =NW,
                text = 'Le moulin de la galette',
                font =('Dom Casual BT', 20), fill ='maroon')

def tabDegas(self):
    self.can.create_text(10, 160, anchor =NW, text = 'Danseuses au repos',
                font =('President', 20), fill ='purple')
```

Analyse du script :

Vous pouvez réaliser aisément des menus en cascade, en enchaînant des sous-menus les uns aux autres jusqu'à un niveau quelconque (il vous est cependant déconseillé d'aller au-delà de 5 niveaux successifs : vos utilisateurs s'y perdraient).

Un sous-menu est défini comme un menu "esclave" du menu de niveau précédent (dans notre exemple, **me2** est défini comme un menu "esclave" de **me1**). L'intégration est assurée ensuite à l'aide de la méthode **add_cascade()**.

L'un des items est désactivé (option **state** = DISABLED). L'exemple suivant vous montrera comment vous pouvez activer ou désactiver à volonté des items, par programme.

14.5.4 Ajout de la rubrique "Options" :

La définition de cette rubrique est un peu plus compliquée, parce que nous allons y intégrer l'utilisation de variables internes à *Tkinter*.

Les fonctionnalités de ce menu sont cependant beaucoup plus élaborées : les options ajoutées permettent en effet d'activer ou de désactiver à volonté les rubriques "Musiciens" et "Peintres", et vous pouvez également modifier à volonté l'aspect de la barre de menus elle-même.



Veuillez donc ajouter les lignes suivantes dans le constructeur de la classe **MenuBar()** :

```
        ##### Menu <Options> #####
        optMenu = Menubutton(self, text ='Options')
        optMenu.pack(side =LEFT, padx ='3')
        # Variables Tkinter :
        self.relief = IntVar()
        self.actPein = IntVar()
        self.actMusi = IntVar()
        # Partie "déroulante" du menu :
        self.mo = Menu(optMenu)
        self.mo.add_command(label = 'Activer :', foreground ='blue')
        self.mo.add_checkbutton(label ='musiciens',
                   command = self.choixActifs, variable =self.actMusi)
        self.mo.add_checkbutton(label ='peintres',
                   command = self.choixActifs, variable =self.actPein)
        self.mo.add_separator()
        self.mo.add_command(label = 'Relief :', foreground ='blue')
        for (v, lab) in [(0,'aucun'), (1,'sorti'), (2,'rentré'),
                         (3,'sillon'), (4,'crête'), (5,'bordure')]:
            self.mo.add_radiobutton(label =lab, variable =self.relief,
                                    value =v, command =self.reliefBarre)
        # Intégration du menu :
        optMenu.configure(menu = self.mo)
```

... ainsi que les définitions de méthodes suivantes (toujours dans la classe **MenuBar()**) :

```
    def reliefBarre(self):
        choix = self.relief.get()
        self.configure(relief =[FLAT,RAISED,SUNKEN,GROOVE,RIDGE,SOLID][choix])

    def choixActifs(self):
        p = self.actPein.get()
        m = self.actMusi.get()
        self.pein.configure(state =[DISABLED, NORMAL][p])
        self.musi.configure(state =[DISABLED, NORMAL][m])
```

Analyse du script

a) Menu avec "cases à cocher"

Notre nouveau menu déroulant comporte deux parties. Afin de bien les mettre en évidence, nous avons inséré une ligne de séparation ainsi que deux "faux items" ("Activer :" et "Relief :") qui servent simplement de titres. Nous faisons apparaître ceux-ci en couleur pour que l'utilisateur ne les confonde pas avec de véritables commandes.

Les items de la première partie sont dotées de "cases à cocher". Lorsque l'utilisateur effectue un clic de souris sur l'un ou l'autre de ces items, les options correspondantes sont activées ou désactivées, et ces états "actif / inactif" sont affichés sous la forme d'une coche. Les instructions qui servent à mettre en place ce type de rubrique sont assez explicites. Elles présentent en effet ces items comme des widgets de type "chekbutton" :

```
self.mo.add_checkbutton(label = 'musiciens', command = choixActifs,
                        variable = mbu.me1.music)
```

Il est important de comprendre ici que ce type de widget comporte nécessairement une variable interne, destinée à mémoriser l'état "actif / inactif" du widget. Cette variable ne peut pas être une variable Python ordinaire, parce que les classes de la bibliothèque *Tkinter* sont écrites dans un autre langage. Et par conséquent, on ne pourra accéder à une telle variable interne qu'à travers une interface. Cette interface, appelée "*variable Tkinter*", est en fait un ***objet***, que l'on crée à partir d'une ***classe*** particulière, qui fait partie du module *Tkinter* au même titre que les classes de widgets. L'utilisation de ces "objets-variables" est relativement simple :

- La classe **IntVar()** permet de créer des objets équivalents à des variables de type entier.
  On commence donc par créer un ou plusieurs de ces objets-variables, que l'on mémorise dans notre exemple comme de nouveaux attribiuts d'instance :
  ```
  self.actMusi =IntVar()
  ```
  Après cette affectation, l'objet référencé dans **self.actMusi** contient désormais l'équivalent d'une variable de type entier, dans un format spécifique à *Tkinter*.
- Ensuite, on associe l'option "variable" de l'objet **checkbutton** à la *variable Tkinter* ainsi définie :
  ```
  self.mo.add_checkbutton(label ='musiciens', variable =self.actMusi)
  ```
- Il est nécessaire de procéder ainsi en deux étapes, parce que *Tkinter* ne peut pas directement assigner des valeurs aux variables Python. Pour une raison similaire, il n'est pas possible à Python de lire directement le contenu d'une *variable Tkinter.* Il faut utiliser pour cela une méthode spécifique de cette classe d'objets : la méthode **get()** [54] :
  ```
  m = self.actMusi.get()
  ```
  Dans cette instruction, nous affectons à **m** (variable ordinaire de Python) le contenu d'une *variable Tkinter* (laquelle est elle-même associée à un widget bien déterminé).

Tout ce qui précède peut vous paraître un peu compliqué. Considérez simplement qu'il s'agit de votre première rencontre avec les problèmes d'***interfaçage*** entre deux langages de programmation différents, utilisés ensemble dans un projet composite.

54 Pour ***écrire*** dans une variable *Tkinter*, il faudrait utiliser la méthode **set()** . Exemple :
```
self.actMusi.set(45)
```

b) Menu avec choix exclusifs

La deuxième partie du menu "Options" permet à l'utilisateur de choisir l'aspect que prendra la barre de menus, parmi six possibilités. Il va de soi que l'on ne peut activer qu'une seule de ces possibilités à la fois. Pour mettre en place ce genre de fonctionnalité, on fait classiquement appel appel à des widgets de type "boutons radio". La caractéristique essentielle de ces widgets est que plusieurs d'entre eux doivent être associés à une seule et même *variable Tkinter*. A chaque bouton radio correspond alors une valeur particulière, et c'est cette valeur qui est affectée à la variable lorsque l'utilisateur sélectionne le bouton.

Ainsi, l'instruction :

```
self.mo.add_radiobutton(label ='sillon', variable =self.relief,
                        value =3, command =self.reliefBarre)
```

configure un item du menu "Options" de telle manière qu'il se comporte comme un bouton radio.

Lorsque l'utilisateur sélectionne cet item, la valeur 3 est affectée à la *variable Tkinter* **self.relief** (celle-ci étant désignée à l'aide de l'option **variable** du widget), et un appel est lancé en direction de la méthode **reliefBarre()**. Celle-ci récupère alors la valeur mémorisée dans la *variable Tkinter* pour effectuer son travail.

Dans le contexte particulier de ce menu, nous souhaitons proposer 6 possibilités différentes à l'utilisateur. Il nous faut donc six "boutons radio", pour lesquels nous pourrions encoder six instructions similaires à celle que nous avons reproduite ci-dessus, chacune d'elles ne différant des cinq autres que par ses options **value** et **label**. Dans une situation de ce genre, la bonne pratique de programmation consiste à placer les valeurs de ces options dans une liste, et à parcourir ensuite cette liste à l'aide d'une boucle **for,** afin d'instancier les widgets avec une instruction commune :

```
for (v, lab) in [(0,'aucun'), (1,'sorti'), (2,'rentré'),
                 (3,'sillon'), (4,'crête'), (5,'bordure')]:
    self.mo.add_radiobutton(label =lab, variable =self.relief,
                            value =v, command =self.reliefBarre)
```

La liste utilisée est une liste de 6 tuples (valeur, libellé). A chacune des 6 itérations de la boucle, un nouvel item *radiobutton* est instancié, dont les options **label** et **value** sont extraites de la liste par l'intermédiaire des variables **lab** et **v**.

Dans vos projets personnels, il vous arrivera fréquemment de constater que vous pouvez ainsi remplacer des suites d'instructions similaires, par une structure de programmation plus compacte (en général, la combinaison d'une liste et d'une boucle, comme dans l'exemple ci-dessus).

Vous découvrirez petit à petit encore d'autres techniques pour alléger votre code : nous en fournissons encore un exemple dans le paragraphe suivant. Tâchez cependant de garder à l'esprit cette règle essentielle, qu'un bon programme doit avant tout rester ***lisible*** et ***commenté***.

c) Contrôle du flux d'exécution à l'aide d'une liste

Veuillez à présent considérer la définition de la méthode **reliefBarre()** :

A la première ligne, la méthode **get()** nous permet de récupérer l'état d'une *variable Tkinter* qui contient le numéro du choix opéré par l'utilisateur dans le sous-menu "Relief :".

A la seconde ligne, nous utilisons le contenu de la variable **choix** pour extraire d'une liste de six éléments celui qui nous intéresse. Par exemple, si **choix** contient la valeur 2, c'est l'option SUNKEN qui sera utilisée pour reconfigurer le widget.

La variable **choix** est donc utilisée ici comme un index, servant à désigner un élément de la liste. En lieu et place de cette construction compacte, nous aurions pu programmer une série de tests conditionnels, comme par exemple :

```
if choix ==0:
    self.configure(relief =FLAT)
elif choix ==1:
    self.configure(relief =RAISED)
elif choix ==2:
    self.configure(relief =SUNKEN)
...
etc.
```

D'un point de vue strictement fonctionnel, le résultat serait exactement le même. Vous admettrez cependant que la construction que nous avons choisie est d'autant plus efficiente, que le nombre de possibilités de choix est élevé. Imaginez par exemple que l'un de vos programmes personnels doive effectuer une sélection dans un très grand nombre d'éléments : avec une construction du type ci-dessus, vous seriez peut-être amené à encoder plusieurs pages de "**elif**" !

Nous utilisons encore la même technique dans la méthode **choixActifs()**. Ainsi l'instruction :

```
self.pein.configure(state =[DISABLED, NORMAL][p])
```

utilise le contenu de la variable **p** comme index pour désigner lequel des deux états DISABLED, NORMAL doit être sélectionné pour reconfigurer le menu "Peintres".

Lorsqu'elle est appelée, la méthode **choixActifs()** reconfigure donc les deux rubriques "Peintres" et "Musiciens" de la barre de menus, pour les faire apparaître "normales" ou "désactivées" en fonction de l'état des variables **m** et **p**, lesquelles sont elles-mêmes le reflet de variables *Tkinter*.

Ces variables intermédiaires **m** et **p** ne servent en fait qu'à clarifier le script. Il serait en effet parfaitement possible de les éliminer, et de rendre le script encore plus compact, en utilisant la composition d'instructions. On pourrait par exemple remplacer les deux instructions :

```
m = self.actMusi.get()
self.musi.configure(state =[DISABLED, NORMAL][m])
```

par une seule, telle que :

```
self.musi.configure(state =[DISABLED, NORMAL][self.actMusi.get()])
```

Notez cependant que ce que l'on gagne en compacité se paie d'une certaine perte de lisibilité.

d) Pré-sélection d'une rubrique

Pour terminer cet exercice, voyons encore comment vous pouvez déterminer à l'avance certaines sélections, ou bien les modifier par programme.

Veuillez donc ajouter l'instruction suivante dans le constructeur de la classe **Application()** (juste avant l'instruction **self.pack()** ,par exemple) :

```
mBar.mo.invoke(2)
```

Lorsque vous exécutez le script ainsi modifié, vous constatez qu'au départ la rubrique "Musiciens" de la barre de menus est active, alors que la rubrique "Peintres" ne l'est pas. Programmées comme elles le sont, ces deux rubriques devraient être actives toutes deux par défaut. Et c'est effectivement ce qui se passe si nous supprimons l'instruction :

```
mBar.mo.invoke(2)
```

Nous vous avons suggéré d'ajouter cette instruction au script, pour vous montrer comment vous pouvez effectuer ***par programme*** la même opération que celle que l'on obtient normalement avec un clic de souris.

L'instruction ci-dessus ***invoque*** le widget **mBar.mo** en actionnant la commande associée au ***deuxième*** item de ce widget. En consultant le listing, vous pouvez vérifier que ce deuxième item est bien l'objet de type *checkbutton* qui active/désactive le menu "Peintres" (Rappelons encore une fois que l'on numérote toujours à partir de zéro).

Au démarrage du programme, tout se passe donc comme si l'utilisateur effectuait tout de suite un premier clic sur la rubrique "Peintres" du menu "Options", ce qui a pour effet de désactiver le menu correspondant.

14.6 Exemple de programme concret : jeu de Ping

Dans les pages qui suivent, vous trouverez le script correspondant à un petit programme complet. Ce programme vous est fourni à titre d'exemple de ce que vous pouvez envisager de développer vous-même comme projet personnel de synthèse. Il vous montre également comment vous pouvez utiliser plusieurs classes afin de construire un script bien structuré.

14.6.1 Principe

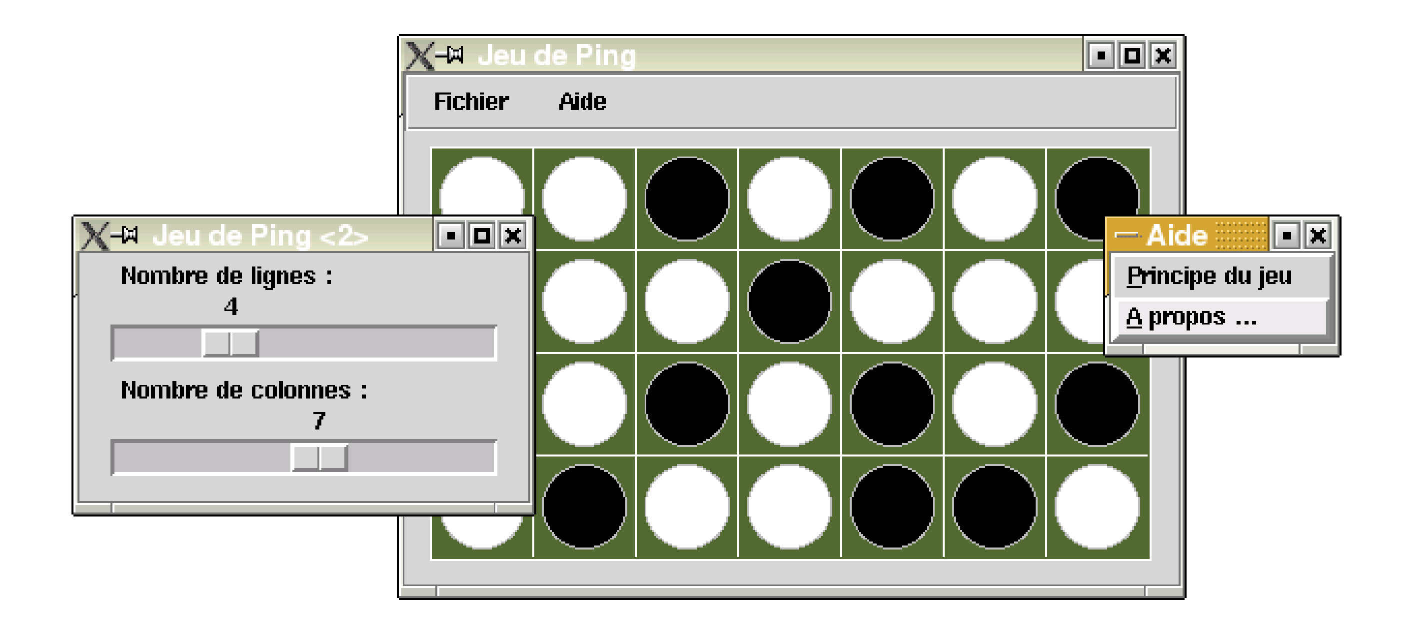
Le "jeu" mis en oeuvre ici est plutôt une sorte d'exercice mathématique. Il se joue sur un panneau ou est représenté un quadrillage de dimensions variables, dont toutes les cases sont occupées par des pions. Ces pions possèdent chacun une face blanche et une face noire (comme les pions du jeu Othello/Reversi), et au début de l'exercice ils présentent tous leur face blanche par-dessus.

Lorsque l'on clique sur un pion à l'aide de la souris, les 8 pions adjacents se retournent.

Le jeu consiste alors à essayer de retourner tous les pions, en cliquant sur certains d'entre eux.

L'exercice est très facile avec une grille de 2 x 2 cases (il suffit de cliquer sur chacun des 4 pions). Il devient plus difficile avec des grilles plus grandes, et est même tout à fait impossible avec certaines d'entre elles. A vous de déterminer lesquelles !

(Ne négligez pas d'étudier le cas des grilles 1 x n).



Note : Vous trouverez la discussion complète du jeu de Ping, sa théorie et ses extensions, dans la revue "Pour la science" n° 298 - Août 2002, pages 98 à 102.

14.6.2 Programmation

Lorsque vous développez un projet logiciel, veillez toujours à faire l'effort de décrire votre démarche le plus clairement possible. Commencez par établir un cahier des charges détaillé, et ne négligez pas de commenter ensuite très soigneusement votre code, au fur et à mesure de son élaboration (et non après coup !).

En procédant ainsi, vous vous forcez vous-même à exprimer ce que vous souhaitez que la machine fasse, ce qui vous aide à analyser les problèmes et à structurer convenablement votre code.

Cahier des charges du logiciel à développer

- L'application sera construite sur la base d'une fenêtre principale comportant le panneau de jeu et une barre de menus.
- L'ensemble devra être extensible à volonté par l'utilisateur, les cases du panneau devant cependant rester carrées.
- Les options du menu permettront de :
  - choisir les dimensions de la grille (en nombre de cases)
  - réinitialiser le jeu (c.à.d. disposer tous les pions avec leur face blanche au-dessus)
  - afficher le principe du jeu dans une fenêtre d'aide
  - terminer.(fermer l'application)
- La programmation fera appel à trois classes :
  - une classe principale
  - une classe pour la barre de menus
  - une classe pour le panneau de jeu
- Le panneau de jeu sera dessiné dans un canevas, lui-même installé dans un cadre (*frame*). En fonction des redimensionnements opérés par l'utilisateur, le cadre occupera à chaque fois toute la place disponible : il se présente donc au programmeur comme un rectangle quelconque, dont les dimensions doivent servir de base au calcul des dimensions de la grille à dessiner.
- Puisque les cases de cette grille doivent rester carrées, il est facile de commencer par calculer leur taille maximale, puis d'établir les dimensions du canevas en fonction de celle-ci.

```
###########################################
#  Jeu de ping                            #
#  Références : Voir article de la revue  #
#  <Pour la science>, Aout 2002           #
#                                         #
# (C) Gérard Swinnen (Verviers, Belgique) #
# http://www.ulg.ac.be/cifen/inforef/swi  #
#                                         #
#  Version du 29/09/2002 - Licence : GPL  #
###########################################

from Tkinter import *

class MenuBar(Frame):
    """Barre de menus déroulants"""
    def __init__(self, boss =None):
        Frame.__init__(self, borderwidth =2, relief =GROOVE)
        ##### Menu <Fichier> #####
        fileMenu = Menubutton(self, text ='Fichier')
        fileMenu.pack(side =LEFT, padx =5)
        me1 = Menu(fileMenu)
        me1.add_command(label ='Options', underline =0,
                        command = boss.options)
        me1.add_command(label ='Restart', underline =0,
                        command = boss.reset)
        me1.add_command(label ='Terminer', underline =0,
                        command = boss.quit)
        fileMenu.configure(menu = me1)

        ##### Menu <Aide> #####
        helpMenu = Menubutton(self, text ='Aide')
        helpMenu.pack(side =LEFT, padx =5)
        me1 = Menu(helpMenu)
        me1.add_command(label ='Principe du jeu', underline =0,
                        command = boss.principe)
        me1.add_command(label ='A propos ...', underline =0,
                        command = boss.aPropos)
        helpMenu.configure(menu = me1)

class Panneau(Frame):
    """Panneau de jeu (grille de n x m cases)"""
    def __init__(self, boss =None):
        # Ce panneau de jeu est constitué d'un cadre redimensionnable
        # contenant lui-même un canevas. A chaque redimensionnement du
        # cadre, on calcule la plus grande taille possible pour les
        # cases (carrées) de la grille, et on adapte les dimensions du
        # canevas en conséquence.
        Frame.__init__(self)
        self.nlig, self.ncol = 4, 4         # Grille initiale = 4 x 4
        # Liaison de l'événement <resize> à un gestionnaire approprié :
        self.bind("<Configure>", self.redim)
        # Canevas :
        self.can =Canvas(self, bg ="dark olive green", borderwidth =0,
                         highlightthickness =1, highlightbackground ="white")
        # Liaison de l'événement <clic de souris> à son gestionnaire :
        self.can.bind("<Button-1>", self.clic)
        self.can.pack()
        self.initJeu()
```

```
    def initJeu(self):
        "Initialisation de la liste mémorisant l'état du jeu"
        self.etat =[]                # construction d'une liste de listes
        for i in range(12):          # (équivalente à un tableau
            self.etat.append([0]*12) #  de 12 lignes x 12 colonnes)

    def redim(self, event):
        "Opérations effectuées à chaque redimensionnement"
        # Les propriétés associées à l'événement de reconfiguration
        # contiennent les nouvelles dimensions du cadre :
        self.width, self.height = event.width -4, event.height -4
        # La différence de 4 pixels sert à compenser l'épaisseur
        # de la 'highlightbordure" entourant le canevas)
        self.traceGrille()

    def traceGrille(self):
        "Dessin de la grille, en fonction des options & dimensions"
        # largeur et hauteur maximales possibles pour les cases :
        lmax = self.width/self.ncol
        hmax = self.height/self.nlig
        # Le coté d'une case sera égal à la plus petite de ces dimensions :
        self.cote = min(lmax, hmax)
        # -> établissement de nouvelles dimensions pour le canevas :
        larg, haut = self.cote*self.ncol, self.cote*self.nlig
        self.can.configure(width =larg, height =haut)
        # Tracé de la grille :
        self.can.delete(ALL)                # Effacement dessins antérieurs
        s =self.cote
        for l in range(self.nlig -1):       # lignes horizontales
            self.can.create_line(0, s, larg, s, fill="white")
            s +=self.cote
        s =self.cote
        for c in range(self.ncol -1):       # lignes verticales
            self.can.create_line(s, 0, s, haut, fill ="white")
            s +=self.cote
        # Tracé de tous les pions, blancs ou noirs suivant l'état du jeu :
        for l in range(self.nlig):
            for c in range(self.ncol):
                x1 = c *self.cote +5            # taille des pions =
                x2 = (c +1)*self.cote -5        # taille de la case -10
                y1 = l *self.cote +5            #
                y2 = (l +1)*self.cote -5
                coul =["white","black"][self.etat[l][c]]
                self.can.create_oval(x1, y1, x2, y2, outline ="grey",
                                     width =1, fill =coul)
```

```
    def clic(self, event):
        "Gestion du clic de souris : retournement des pions"
        # On commence par déterminer la ligne et la colonne :
        lig, col = event.y/self.cote, event.x/self.cote
        # On traite ensuite les 8 cases adjacentes :
        for l in range(lig -1, lig+2):
            if l <0 or l >= self.nlig:
                continue
            for c in range(col -1, col +2):
                if c <0 or c >= self.ncol:
                    continue
                if l ==lig and c ==col:
                    continue
                # Retournement du pion par inversion logique :
                self.etat[l][c] = not (self.etat[l][c])
        self.traceGrille()


class Ping(Frame):
    """corps principal du programme"""
    def __init__(self):
        Frame.__init__(self)
        self.master.geometry("400x300")
        self.master.title(" Jeu de Ping")

        self.mbar = MenuBar(self)
        self.mbar.pack(side =TOP, expand =NO, fill =X)

        self.jeu =Panneau(self)
        self.jeu.pack(expand =YES, fill=BOTH, padx =8, pady =8)

        self.pack()

    def options(self):
        "Choix du nombre de lignes et de colonnes pour la grille"
        opt =Toplevel(self)
        curL =Scale(opt, length =200, label ="Nombre de lignes :",
              orient =HORIZONTAL,
              from_ =1, to =12, command =self.majLignes)
        curL.set(self.jeu.nlig)     # position initiale du curseur
        curL.pack()
        curH =Scale(opt, length =200, label ="Nombre de colonnes :",
              orient =HORIZONTAL,
              from_ =1, to =12, command =self.majColonnes)
        curH.set(self.jeu.ncol)
        curH.pack()

    def majColonnes(self, n):
        self.jeu.ncol = int(n)
        self.jeu.traceGrille()

    def majLignes(self, n):
        self.jeu.nlig = int(n)
        self.jeu.traceGrille()
```

```
    def reset(self):
        self.jeu.initJeu()
        self.jeu.traceGrille()

    def principe(self):
        "Fenêtre-message contenant la description sommaire du principe du jeu"
        msg =Toplevel(self)
        Message(msg, bg ="navy", fg ="ivory", width =400,
            font ="Helvetica 10 bold",
            text ="Les pions de ce jeu possèdent chacun une face blanche et "\
            "une face noire. Lorsque l'on clique sur un pion, les 8 "\
            "pions adjacents se retournent.\nLe jeu consiste a essayer "\
            "de les retouner tous.\n\nSi l'exercice se révèle très facile "\
            "avec une grille de 2 x 2 cases. Il devient plus difficile avec "\
            "des grilles plus grandes. Il est même tout à fait impossible "\
            "avec certaines grilles.\nA vous de déterminer lesquelles !\n\n"\
            "Réf : revue 'Pour la Science' - Aout 2002")\
            .pack(padx =10, pady =10)

    def aPropos(self):
        "Fenêtre-message indiquant l'auteur et le type de licence"
        msg =Toplevel(self)
        Message(msg, width =200, aspect =100, justify =CENTER,
            text ="Jeu de Ping\n\n(C) Gérard Swinnen, Aout 2002.\n"\
            "Licence = GPL").pack(padx =10, pady =10)

if __name__ == '__main__':
    Ping().mainloop()
```

Note : *Si vous souhaitez expérimenter ce petit programme sans avoir à le réécrire, vous pouvez trouver son code source à l'adresse : http://www.ulg.ac.be/cifen/inforef/swi/python.htm*

Chapitre 15 : CGI et Python

15.1 Pages Web interactives

Vous avez certainement déjà appris un grand nombre de choses concernant la rédaction de pages Web dans d'autres cours. Vous savez que ces pages sont des documents au format HTML, que l'on peut consulter via un réseau (intranet ou internet) à l'aide d'un logiciel appelé *browser Web* ou *navigateur* (Netscape, Konqueror, Internet explorer, ...).

Les pages HTML sont installées dans les répertoires publics d'un autre ordinateur où fonctionne en permanence un logiciel appelé *serveur Web* (Apache, IIS, Zope, ...). Lorsqu'une connexion a été établie entre cet ordinateur et le vôtre, votre logiciel navigateur peut dialoguer avec le logiciel serveur (par l'intermédiaire de toute une série de dispositifs matériels et logiciels dont nous ne traiterons pas ici : lignes téléphoniques, routeurs, caches, protocoles de communication ...).

Le protocole HTML autorise l'échange de données dans les deux sens, mais dans la grande majorité des cas, le transfert d'informations n'a pratiquement lieu que dans un seul, à savoir du serveur vers le navigateur : des textes, des images, des fichiers divers lui sont expédiés en grand nombre (ce sont les pages consultées) ; en revanche, le navigateur n'envoie guère au serveur que de toutes petites quantités d'information : essentiellement les adresses URL des pages que l'internaute désire consulter.

Vous savez cependant qu'il existe des sites Web où vous êtes invité à fournir vous-même des quantités d'information plus importantes : vos références personnelles pour l'inscription à un club ou la réservation d'une chambre d'hôtel, votre numéro de carte de crédit pour la commande d'un article sur un site de commerce électronique, votre avis ou vos suggestions, etc.

Dans un cas comme ceux-là, vous vous doutez bien que l'information transmise doit être prise en charge, du côté du serveur, par un programme spécifique. Il faut donc que les pages Web destinées à accueillir cette information soient dotées d'un mécanisme assurant son transfert vers le logiciel destiné à la traiter. Il faudra également que ce logiciel puisse lui-même transmettre en retour une information au serveur, afin que celui-ci puisse présenter le résultat de l'opération à l'internaute, sous la forme d'une nouvelle page Web.

Le but du présent chapitre est de vous expliquer comment vous pouvez vous servir de vos compétences de programmeur Python pour ajouter une telle interactivité à un site Web, en y intégrant de véritables applications.

Remarque importante : Ce que nous allons expliquer dans les paragraphes qui suivent sera directement fonctionnel sur l'intranet de votre école (à la condition toutefois que l'administrateur de votre intranet scolaire ait configuré son serveur comme nous l'expliquons plus loin). En ce qui concerne l'internet, les choses sont un peu plus compliquées. Il va de soi que l'installation de logiciels sur un ordinateur serveur relié à l'internet ne peut se faire qu'avec l'accord de son propriétaire. Si un fournisseur d'accès à l'internet a mis a votre disposition un certain espace où vous êtes autorisé à installer des pages Web "ordinaires" (c.à.d. de simples documents à consulter), cela ne signifie pas pour autant que vous pourrez y faire fonctionner des scripts Python. Pour que cela puisse marcher, vous devrez demander une autorisation et un certain nombre de renseignements à votre fournisseur d'accès. Il faudra en particulier lui demander si vous pouvez activer des scripts CGI écrits en Python à partir de vos pages, et dans quel(s) répertoire(s) vous pouvez les installer.

15.2 L'interface CGI

L'interface CGI (pour *Common Gateway Interface*) est un composant de la plupart des logiciels serveurs de pages Web. Il s'agit d'une passerelle qui leur permet de communiquer avec d'autres logiciels tournant sur le même ordinateur. Avec CGI, vous pouvez écrire des scripts dans différents langages (Perl, C, Tcl, Python ...).

Plutôt que de limiter le Web à des documents écrits à l'avance, CGI permet de générer des pages Web sur le champ, en fonction des données que fournit l'internaute par l'intermédiaire de son logiciel de navigation. Vous pouvez utiliser les scripts CGI pour créer une large palette d'applications : des services d'inscription en ligne, des outils de recherche dans des bases de données, des instruments de sondage d'opinions, des jeux, etc.

L'apprentissage de la programmation CGI peut faire l'objet de manuels entiers (Veuillez consulter votre professeur à ce sujet). Nous nous limiterons dans ce cours aux principes de base.

15.3 Une interaction CGI rudimentaire

Notre premier exemple sera constitué d'une page Web extrêmement simple. Nous n'y placerons qu'un seul élément d'interactivité, à savoir un unique bouton. Ce bouton servira à lancer l'exécution d'un petit programme que nous décrirons par après.

Veuillez donc encoder le document HTML ci-dessous à l'aide d'un éditeur quelconque (n'encodez pas les numéros de lignes, ils ne sont là que pour faciliter les explications qui suivent) :

```
1.   <HTML>
2.   <HEAD><TITLE>Exercice avec Python</TITLE></HEAD>
3.   <BODY>
4.
5.   <DIV ALIGN="center">
6.   <IMG SRC="penguin.gif">
7.   <H2>Page Web interactive</H2>
8.   <P>Cette page est associée à un script Python</P>
9.
10.  <FORM ACTION="http://Serveur/cgi-bin/input_query.py" METHOD="post">
11.  <INPUT TYPE="submit" NAME="send" VALUE="Exécuter le script">
12.  </FORM>
13.
14.  </DIV></BODY></HTML>
```

Vous savez certainement déjà que les balises initiales <HTML>, <HEAD>, <TITLE>, <BODY>, ainsi que les balises finales correspondantes, sont communes à tous les documents HTML. Nous ne détaillerons donc pas leur rôle ici.

La balise <DIV> utilisée à la ligne 5 sert habituellement à diviser un document HTML en sections distinctes. Nous l'utilisons ici pour définir une section dans laquelle tous les éléments seront centrés (horizontalement) sur la page.

A la ligne 6, nous insérons une petite image.

La ligne 7 définit une ligne te texte comme étant un titre de 2e importance.

La ligne 8 est un paragraphe ordinaire.

Les lignes 10 à 12 contiennent le code important (pour ce qui nous occupe ici). Les balises <FORM> et </FORM> définissent en effet un **formulaire**, c.à.d. une portion de page Web susceptible de contenir divers widgets à l'aide desquels l'internaute pourra exercer une certaine activité : champs d'entrée, boutons, cases à cocher, boutons radio, etc.

La balise FORM doit contenir deux indications essentielles : l'**action** à accomplir lorsque le formulaire sera expédié (il s'agit en fait de fournir ici l'adresse URL du logiciel à invoquer pour traiter les données transmises), et la **méthode** à utiliser pour transmettre l'information (en ce qui nous concerne, ce sera toujours la méthode "post").

Dans notre exemple, le logiciel que nous voulons invoquer est un script Python nommé **input_query.py** qui est situé dans un répertoire particulier du serveur d'intranet. Sur de nombreux serveurs, ce répertoire s'appelle souvent *cgi-bin* , par pure convention. Si le serveur Web de votre intranet scolaire a été configuré comme nous le suggérons dans l'une des annexes de ces notes, vous devriez pouvoir installer vos scripts Python dans le répertoire que l'on vous a déjà attribué spécifiquement pour y placer vos pages Web personnelles.

Vous devrez donc modifier la ligne 10 de notre exemple, en remplaçant l'adresse *http://Serveur/cgi-bin/input_query.py* par ce que votre professeur vous indiquera[55].

La ligne 11 contient la balise qui définit un widget de type "bouton d'envoi" (balise <INPUT TYPE="submit">). Le texte qui doit apparaître sur le bouton est précisé par l'attribut VALUE ="texte". L'indication NAME est facultative dans le cas présent. Elle mentionne le nom du widget lui-même (au cas où le logiciel destinataire en aurait besoin).

Lorsque vous aurez terminé l'encodage de ce document, sauvegardez-le dans le répertoire que l'on vous a attribué spécifiquement pour y placer vos pages, sous un nom quelconque, mais de préférence avec l'extension .html ou .htm (par exemple : essai.html).

Le script Python **input_query.py** est détaillé ci-dessous. Comme déjà signalé plus haut, vous pouvez installer ce script dans le même répertoire que votre document HTML initial :

```
#! /usr/bin/python

# Affichage d'un formulaire HTML simplifié :
print "Content-Type: text/html\n"
print """
<H3><FONT COLOR="Royal blue">
Page web produite par un script Python
</FONT></H3>

<FORM ACTION="print_result.py" METHOD="post">
<P>Veuillez entrer votre nom dans le champ ci-dessous, s.v.p. :</P>
<P><INPUT NAME="visiteur" SIZE=20 MAXLENGTH=20 TYPE="text"></P>
<P>Veuillez également me fournir une phrase quelconque :</P>
<TEXTAREA NAME="phrase" ROWS=2 COLS=50>Mississippi</TEXTAREA>
<P>J'utiliserai cette phrase pour établir un histogramme.</P>
<INPUT TYPE="submit" NAME="send" VALUE="Action">
</FORM>
"""
```

55 Par exemple : http://192.168.0.100/cgi/Classe6A/Dupont/input_query.py – Veuillez consulter les annexes pour explications complémentaires.

Ce script ne fait rien d'autre que d'afficher une nouvelle page Web, laquelle contient encore une fois un formulaire, mais celui-ci nettement plus élaboré que le précédent.

La première ligne est absolument nécessaire : elle indique à l'interface CGI qu'il faut lancer l'interpréteur Python pour pouvoir exécuter le script. La seconde ligne est un simple commentaire.

La ligne 4 est indispensable. Elle permet à l'interpréteur Python d'initialiser un véritable document HTML qui sera transmis au serveur Web. Celui-ci pourra à son tour le réexpédier au logiciel navigateur de l'internaute, et celui-ci le verra donc s'afficher dans la fenêtre de navigation.

La suite est du pur code HTML, traité par Python comme une simple chaîne de caractères que l'on affiche à l'aide de l'instruction **print**. Pour pouvoir y insérer tout ce que nous voulons, y compris les sauts à la ligne, les apostrophes, les guillemets, etc., nous délimitons cette chaîne de caractères à l'aide de "triples guillemets" (Rappelons ici que les sauts à la ligne sont complètement ignorés en HTML : nous pouvons donc en utiliser autant que nous voulons pour "aérer" notre code et le rendre plus lisible).

15.4 Un formulaire HTML pour l'acquisition des données

Analysons à présent le code HTML lui-même. Nous y trouvons essentiellement un nouveau formulaire, qui comporte plusieurs paragraphes, parmi lesquels on peut reconnaître quelques widgets. La ligne 10 indique le nom du script CGI auquel les données du formulaire seront transmises : il s'agira bien évidemment d'un autre script Python.

A la ligne 12, on trouve la définition d'un widget de type "champ d'entrée" (Balise INPUT, avec TYPE="text"). L'utilisateur est invité à y encoder son nom. Le paramètre MAXLENGTH définit une longueur maximale pour la chaîne de caractères qui sera entrée ici (20caractères, en l'occurrence). Le paramètre SIZE définit la taille du champ tel qu'il doit apparaître à l'écran, et le paramètre NAME est le nom que nous choisissons pour la variable destinée à mémoriser la chaîne de caractères attendue.

Un second champ d'entrée un peu différent est défini à la ligne 14 (balise TEXTAREA). Il s'agit d'un réceptacle plus vaste, destiné à accueillir des textes de plusieurs lignes. (Ce champ est automatiquement pourvu d'ascenseurs si le texte à insérer se révèle trop volumineux). Ses paramètres ROWS et COLS sont assez explicites. Entre les balises initiale et finale, on peut insérer un texte par défaut ("Mississippi" dans notre exemple).

Comme dans l'exemple précédent, la ligne 16 contient la définition du bouton qu'il faudra actionner pour transmettre les données au script CGI destinataire, lequel est décrit ci-après.

15.5 Un script CGI pour le traitement des données

Le mécanisme utilisé à l'intérieur d'un script CGI pour réceptionner les données transmises par un formulaire HTML est fort simple, comme vous pouvez l'analyser dans l'exemple ci-dessous :

```
1.    #! /usr/bin/python
2.    # Traitement des données transmises par un formulaire HTML
3.
4.    import cgi                        # Module d'interface avec le serveur Web
5.    form = cgi.FieldStorage()         # Réception de la requête utilisateur :
6.                                      # il s'agit d'une sorte de dictionnaire
7.    if form.has_key("phrase"):        # La clé n'existera pas si le champ
8.       text = form["phrase"].value    # correspondant est resté vide
9.    else:
10.      text ="*** le champ phrase était vide ! ***"
11.
12.   if form.has_key("visiteur"):      # La clé n'existera pas si le champ
13.      nomv = form["visiteur"].value  # correspondant est resté vide
14.   else:
15.      nomv ="mais vous ne m'avez pas indiqué votre nom"
16.
17.   print "Content-Type: text/html\n"
18.   print """
19.   <H3>Merci, %s !</H3>
20.   <H4>La phrase que vous m'avez fournie était : </H4>
21.   <H3><FONT Color="red"> %s </FONT></H3>""" % (nomv, text)
22.
23.   histogr ={}
24.   for c in text:
25.      histogr[c] = histogr.get(c, 0) +1
26.
27.   liste = histogr.items()       # conversion en une liste de tuples
28.   liste.sort()                  # tri de la liste
29.   print "<H4>Fréquence de chaque caractère dans la phrase :</H4>"
30.   for c, f in liste:
31.      print 'le caractère <B>"%s"</B> apparaît %s fois <BR>' % (c, f)
```

Les lignes 4 et 5 sont les plus importantes :

Le module **cgi** importé à la ligne 4 assure la connexion du script Python avec l'interface CGI , laquelle permet de dialoguer avec le serveur Web.

A la ligne 5, la fonction **FieldStorage()** de ce module retourne un objet qui contient l'ensemble des données transmises par le formulaire HTML. Nous plaçons cet objet, lequel est assez semblable à un dictionnaire classique, dans la variable **form**.

Par rapport à un véritable dictionnaire, l'objet placé dans **form** présente la différence essentielle qu'il faudra lui appliquer la méthode **value()** pour en extraire les données. Les autres méthodes applicables aux dictionnaires, telles la méthode **has_key()** , par exemple, peuvent être utilisées de la manière habituelle.

Une caractéristique importante de l'objet dictionnaire retourné par **FieldStorage()** est qu'***il ne possédera aucune clé pour les champs laissés vides*** dans le formulaire HTML correspondant.

Dans notre exemple, le formulaire comporte deux champs d'entrée, auxquels nous avons associé les noms "visiteur" et "phrase". Si ces champs ont effectivement été complétés par l'utilisateur, nous trouverons leurs contenus dans l'objet dictionnaire, aux index "visiteur" et "phrase". Par contre, si l'un ou l'autre de ces champs n'a pas été complété, l'index correspondant n'existera tout simplement pas. Avant toute forme de traitement de valeurs, il est donc indispensable de s'assurer de la présence de chacun des index attendus, et c'est ce que nous faisons aux lignes 7 à 15.

Exercice :

e 179. Pour vérifier ce qui précède, vous pouvez par exemple désactiver (en les transformant en commentaires) les lignes 7, 9, 10, 12, 14 & 15 du script. Si vous testez le fonctionnement de l'ensemble, vous constaterez que tout se passe bien si l'utilisateur complète effectivement les champs qui lui sont proposés. Si l'un des champs est laissé vide, par contre, une erreur se produit.

Note importante : le script étant lancé par l'intermédiaire d'une page Web, les messages d'erreur de Python ne seront pas affichés dans cette page, mais plutôt enregistrés dans le journal des événements du serveur Web. Veuillez consulter votre professeur pour savoir comment vous pouvez accéder à ce journal. De toute manière, attendez-vous à ce que la recherche des erreurs dans un script CGI soit plus ardue que dans une application ordinaire.

Le reste du script est assez classique.

- Aux lignes 17 à 21, nous ne faisons qu'afficher les données transmises par le formulaire. Veuillez noter que les variables **nomv** et **text** doivent exister au préalable, ce qui rend indispensables les lignes 9, 10, 14 & 15.
- Aux lignes 23, 24 & 25, nous nous servons d'un dictionnaire pour construire un histogramme simple, comme nous l'avons expliqué à la page 144.
- A la ligne 27, nous convertissons le dictionnaire résultant en une liste de tuples, pour pouvoir trier celle-ci dans l'ordre alphabétique à la ligne 28.
- La boucle **for** des lignes 30 et 31 se passe de commentaires.

Chapitre 16 : Gestion d'une base de données

16.1 Les bases de données

Il existe de nombreux types de bases de données. On peut par exemple déjà considérer comme une base de données élémentaire, un fichier qui contient une liste de noms et d'adresses.
Si la liste n'est pas trop longue, et si l'on ne souhaite pas pouvoir y effectuer des recherches en fonction de critères complexes, il va de soi que l'on peut accéder à ce type de données en utilisant des instructions simples, telles celles que nous avons abordées page .

La situation se complique cependant très vite si l'on souhaite pouvoir effectuer des sélections et des tris parmi les données, surtout si celles-ci deviennent très nombreuses. La difficulté augmente encore si les données sont répertoriées dans différents ensembles reliés par un certain nombre de relations hiérarchiques, et si plusieurs utilisateurs doivent pouvoir y accéder en parallèle.

Imaginez par exemple que la direction de votre école vous confie la charge de mettre au point un système de bulletins informatisé. En y réfléchissant quelque peu, vous vous rendrez compte rapidement que cela suppose la mise en oeuvre de toute une série de tables différentes : une table des noms d'élèves (laquelle pourra bien entendu contenir aussi d'autres informations spécifiques à ces élèves : adresse, date de naissance, etc.) ; une table contenant la liste des cours (avec le nom du professeur titulaire, le nombre d'heures enseignées par semaine, etc.) ; une table mémorisant les travaux pris en compte pour l'évaluation (avec leur importance, leur date, leur contenu, etc.) ; une table décrivant la manière dont les élèves sont groupés par classes ou par options, les cours suivis par chacun, etc., etc.

Vous comprenez bien que ces différentes tables ne sont pas indépendantes. Les travaux effectués par un même élève sont liés à des cours différents. Pour établir le bulletin de cet élève, il faut donc extraire des données de la table des travaux, bien sûr, mais en relation avec des informations trouvées dans d'autres tables (celles des cours, des classes, des options, etc.)

Nous verrons plus loin comment représenter des tables de données et les relations qui les lient.

16.1.1 SGBDR - Le modèle client/serveur

Les programmes informatiques capables de gérer efficacement de tels ensembles de données complexes sont forcément complexes, eux aussi. On appelle ces programmes des **SGBDR** (***Systèmes de Gestion de Bases de Données Relationnelles***). Il s'agit d'applications informatiques de première importance pour les entreprises. Certaines sont les fleurons de sociétés spécialisées (*IBM®, Oracle®, Microsoft®, Informix®, Sybase®*...) et sont en général vendues à des prix fort élevés. D'autres ont été développées dans des centres de recherche et d'enseignement universitaires (*PostgreSQL©, MySQL©* ...); elles sont alors en général tout à fait gratuites.

Ces systèmes ont chacun leurs spécificités et leurs performances, mais la plupart fonctionnant sur le modèle **client/serveur** : cela signifie que la plus grosse partie de l'application (ainsi que la base de données prise en charge) est installée en un seul endroit, en principe sur une machine puissante (cet ensemble constituant donc le **serveur**), alors que l'autre partie, beaucoup plus simple, est installée sur un nombre indéterminé de postes de travail, et on appelle celles-ci des **clients**.

Les clients sont reliés au serveur, en permanence ou non, par divers procédés et protocoles (éventuellement par l'intermédiaire de l'internet). Chacun d'entre eux peut accéder à une partie plus ou moins importante des données, avec autorisation ou non de modifier certaines d'entre elles, d'en ajouter ou d'en supprimer, en fonction de règles d'accès bien déterminées. (Ces règles sont définies par un **administrateur** de la base de données).

Le serveur et ses clients sont en fait des applications distinctes qui s'échangent des informations. Imaginez par exemple que vous êtes l'un des utilisateurs du système. Pour accéder aux données, vous devez lancer l'exécution d'une application cliente sur un poste de travail quelconque. Dans son processus de démarrage, l'application cliente commence par établir la connexion avec le serveur et la base de données[56]. Lorsque la connexion est établie, l'application cliente peut interroger le serveur en lui envoyant une **requête** sous une forme convenue. Il s'agit par exemple de retrouver une information précise. Le serveur exécute alors la requête en recherchant les données correspondantes dans la base, puis il expédie en retour une certaine **réponse** au client.

Cette réponse peut être l'information demandée, ou encore un message d'erreur en cas d'insuccès.

La communication entre le client et le serveur est donc faite de requêtes et de réponses. Les requêtes sont de véritables instructions expédiées du client au serveur, non seulement pour extraire des données de la base, mais aussi pour en ajouter, en supprimer, en modifier, etc.

16.1.2 Le langage SQL - Gadfly

Étant donnée la diversité des SGBDR existants, on pourrait craindre que chacun d'eux nécessite l'utilisation d'un langage particulier pour les requêtes qu'on lui adresse. En fait, de grands efforts ont été accomplis un peu partout pour la mise au point d'un langage commun, et il existe à présent un standard bien établi : **SQL** (***Structured Query Language***, ou *langage de requêtes structuré*)[57].

Vous aurez probablement l'occasion de rencontrer SQL dans d'autres cours (bureautique, par exemple). Dans le cadre de ces notes de cours concernant l'apprentissage de la programmation avec Python, nous allons nous limiter à la présentation d'un ensemble de modules qui permettent de réaliser aisément un petit SGBDR utilisant SQL, sans autres outils que Python.

Le système que nous allons mettre en oeuvre s'appelle **Gadfly**. Il est entièrement écrit en Python et intègre un large sous-ensemble de commandes SQL. Ses performances ne sont évidemment pas comparables à celles d'un gros SGBDR spécialisé[58], mais elles sont tout à fait excellentes pour la gestion de bases de données modestes. Absolument portable comme Python lui-même, *Gadfly* fonctionnera indifféremment sous Window$, Linux ou M@c. De même, les répertoires contenant des bases de données produites sous *Gadfly* pourront être utilisées sans modification depuis l'un ou l'autre de ces systèmes.

Si vous souhaitez développer une application qui doit gérer des relations relativement complexes dans une petite base de données, le module *Gadfly* peut vous faciliter grandement la tâche.

---

56 il vous faudra certainement entrer quelques informations pour obtenir l'accès : adresse du serveur sur le réseau, nom de la base de données, nom d'utilisateur, mot de passe, ...

57 Quelques variantes subsistent entre différentes implémentations du SQL, pour des requêtes très spécifiques, mais la base reste cependant la même.

58 Gadfly se révèle relativement efficace pour la gestion de bases de données de taille moyenne, en mode mono-utilisateur. Pour gérer de grosses bases de données en mode multi-utilisateur, il faut faire appel à des SGDBR plus ambitieux tels que PostgreSQL, pour lesquels des modules clients Python existent aussi (Pygresql, par ex.).

16.2 Mise en oeuvre d'une base de données simple avec Gadfly

Nous allons d'abord examiner comment mettre en place une application simple, qui fasse office à la fois de serveur et de client sur la même machine. Nous configurerons ensuite cette application en tant que serveur, pour des clients que nous installerons sur d'autres postes.

16.2.1 Création de la base de données

Comme c'est souvent le cas sous Python, il vous suffit d'importer un module – le module gadfly en l'occurrence - pour accéder aux fonctionnalités correspondantes[59].
Vous devez ensuite créer une instance (un objet) de la classe gadfly :

```
import gadfly
connexion = gadfly.gadfly()
```

L'objet **connexion** ainsi créé est votre moteur de base de données local, lequel effectuera la plupart de ses opérations en mémoire vive. Ceci permet une exécution très rapide des requêtes.

Pour créer la base de données proprement dite, il faut employer la méthode **startup** de cet objet :

```
connexion.startup("mydata","\Python20\essais\gadfly")
```

Le premier paramètre transmis, **mydata**, est le nom choisi pour la base de données (vous pouvez évidemment choisir un autre nom !). Le second paramètre est le répertoire où l'on souhaite installer cette base de données. (Ce répertoire doit avoir été créé au préalable, et toute base de données de même nom qui préexisterait dans ce répertoire est écrasée sans avertissement).

Les trois lignes de code que vous venez d'entrer sont suffisantes : vous disposez dès à présent d'une base de données fonctionnelle, dans laquelle vous pouvez créer différentes tables, puis ajouter, supprimer ou modifier des données dans ces tables.

Pour toutes ces opérations, vous allez utiliser le langage SQL.

Afin de pouvoir transmettre vos requêtes SQL à l'objet **connexion** , vous devez cependant mettre en oeuvre un ***curseur***. Il s'agit d'une sorte de tampon mémoire intermédiaire, destiné à mémoriser temporairement les données en cours de traitement, ainsi que les opérations que vous effectuez sur elles, avant leur transfert définitif dans de vrais fichiers. Cette technique permet donc d'annuler si nécessaire une ou plusieurs opérations qui se seraient révélées inadéquates (Vous pouvez en apprendre davantage sur ce concept en consultant l'un des nombreux manuels qui traitent du langage SQL). Veuillez à présent examiner le petit script ci-dessous, et noter que les requêtes SQL sont des chaînes de caractères, prises en charge par la méthode **execute** de l'objet *curseur* :

```
cur = connexion.cursor()
cur.execute("create table membres (age integer, nom varchar, taille float)")
cur.execute("insert into membres(age, nom, taille) values (21,'Dupont',1.83)")
cur.execute("INSERT INTO MEMBRES(AGE, NOM, TAILLE) VALUES (15,'Suleau',1.57)")
cur.execute("Insert Into Membres(Age, Nom, Taille) Values (18,'Forcas',1.69)")
connexion.commit()
```

La première des lignes ci-dessus crée l'objet curseur **cur**. Les chaînes de caractères comprises entre guillemets dans les 4 lignes suivantes contiennent des requêtes SQL très classiques. ***Notez bien que le langage SQL ne tient aucun compte de la casse des caractères*** : vous pouvez encoder vos requêtes SQL indifféremment en majuscules ou en minuscules (ce qui n'est pas le cas pour les instructions Python environnantes, bien entendu !)

59 A la condition que ce module soit installé sur votre système, bien entendu. Le module Gadfly est disponible gratuitement sur l'internet. Voir http://www.chordate.com/kwParsing/gadfly.html
L'installation de ce module est décrite dans l'annexe *** , page ***.

La seconde ligne crée une table nommée **membres**, laquelle contiendra des enregistrements de 3 champs : le champ **age** de type "nombre entier", le champ **nom** de type "chaîne de caractères" (de longueur variable[60]) et le champ **taille**, de type "nombre réel" (à virgule flottante). Le langage SQL autorise en principe d'autres types, mais ils ne sont pas implémentés dans Gadfly.

Les trois lignes qui suivent sont similaires. Nous y avons mélangé majuscules et minuscules pour bien montrer que la casse n'est pas significative en SQL. Ces lignes servent à insérer trois enregistrements dans la table **membres**.

A ce stade des opérations, les enregistrement n'ont pas encore été transférés dans de véritables fichiers sur disque. Il est donc possible de revenir en arrière, comme nous le verrons un peu plus loin. Le transfert sur disque est activé par la méthode **commit()** de la dernière ligne d'instructions.

16.2.2 Connexion à une base de données existante

Supposons qu'à la suite des opérations ci-dessus, nous décidions de terminer le script, ou même d'éteindre l'ordinateur. Comment devrons-nous procéder par la suite pour accéder à nouveau à notre base de données ?

L'accès à une base de données existante ne nécessite que deux lignes de code :

```
import gadfly
connexion = gadfly.gadfly("mydata","\Python20\essais\gadfly")
```

Ces deux lignes suffisent en effet pour transférer en mémoire vive les tables contenues dans les fichiers enregistrés sur disque. La base de données peut désormais être interrogée et modifiée :

```
cur = connexion.cursor()
cur.execute("select * from membres")
print cur.pp()
```

La première de ces trois lignes ouvre un curseur. La requête émise dans la seconde ligne demande la sélection d'un ensemble d'enregistrements, qui seront transférés de la base de données au curseur. Dans le cas présent, la sélection n'en n'est pas vraiment une : on y demande en effet d'extraire **tous** les enregistrements de la table membres (le symbole * est fréquemment utilisé en informatique avec la signification "tout" ou "tous").

La méthode **pp()** utilisée sur le curseur, dans la troisième ligne, provoque un affichage de tout ce qui est contenu dans le curseur sous une forme pré-formatée (les données présentes sont automatiquement disposées en colonnes). "pp" doit en effet être compris comme "*pretty print*".

Si vous préférez contrôler vous-même la mise en page des informations, il vous suffit d'utiliser à sa place la méthode **fetchall()** , laquelle retourne une liste de tuples. Essayez par exemple :

```
for x in cur.fetchall():
    print x, x[0], x[1], x[2]
```

Vous pouvez bien entendu ajouter des enregistrements supplémentaires :

```
cur.execute("Insert Into Membres(Age, Nom, Taille) Values (19,'Ricard',1.75)")
```

Pour modifier un ou plusieurs enregistrements, exécutez une requête du type :

```
cur.execute("update membres set nom ='Gerart' where nom='Ricard'")
```

60 Veuillez noter qu'en SQL, les chaînes de caractères doivent être délimitées par des apostrophes. Si vous souhaitez que la chaîne contienne elle-même une ou plusieurs apostrophes, il vous suffit de doubler celles-ci.

Pour supprimer un ou plusieurs enregistrements, utilisez une requête telle que :

```
cur.execute("delete from membres where nom='Gerart'")
```

Si vous effectuez toutes ces opérations à la ligne de commande de Python, vous pouvez en observer le résultat à tout moment en effectuant un "*pretty print*" comme expliqué plus haut. Étant donné que toutes les modifications apportées au curseur se passent en mémoire vive, rien n'est enregistré définitivement tant que vous n'exécutez pas l'instruction **connexion.commit()**.

Vous pouvez donc annuler toutes les modifications apportées depuis le **commit()** précédent, en refermant la connexion à l'aide de l'instruction :

```
connexion.close()
```

16.2.3 Recherches dans une base de données

e 180. Avant d'aller plus loin, et à titre d'exercice de synthèse, nous allons vous demander de créer entièrement vous-même une base de données "Musique" qui contiendra les deux tables suivantes (Cela représente un certain travail, mais il faut que vous puissiez disposer d'un certain nombre de données pour pouvoir expérimenter les fonctions de recherche et de tri) :

Oeuvres
comp (chaîne)
titre (chaîne)
duree (entier)
interpr (chaîne)

Compositeurs
comp (chaîne)
a_naiss (entier)
a_mort (entier)

Commencez à remplir la table **Compositeurs** avec les données qui suivent (... et profitez de cette occasion pour faire la preuve des compétences que vous maîtrisez déjà, en écrivant un petit script pour vous faciliter l'entrée des informations : une boucle s'impose !)

```
comp          a_naiss   a_mort

Mozart        1756      1791
Beethoven     1770      1827
Handel        1685      1759
Schubert      1797      1828
Vivaldi       1678      1741
Monteverdi    1567      1643
Chopin        1810      1849
Bach          1685      1750
```

Dans la table **Oeuvres**, entrez les données suivantes :

comp	titre	duree	interpr
Vivaldi	**Les quatre saisons**	**20**	**T. Pinnock**
Mozart	**Concerto piano N°12**	**25**	**M. Perahia**
Brahms	**Concerto violon N°2**	**40**	**A. Grumiaux**
Beethoven	**Sonate "au clair de lune"**	**14**	**W. Kempf**
Beethoven	**Sonate "pathétique"**	**17**	**W. Kempf**
Schubert	**Quintette "la truite"**	**39**	**SE of London**
Haydn	**La création**	**109**	**H. Von Karajan**
Chopin	**Concerto piano N°1**	**42**	**M.J. Pires**
Bach	**Toccata & fugue**	**9**	**P. Burmester**
Beethoven	**Concerto piano N°4**	**33**	**M. Pollini**
Mozart	**Symphonie N°40**	**29**	**F. Bruggen**
Mozart	**Concerto piano N°22**	**35**	**S. Richter**
Beethoven	**Concerto piano N°3**	**37**	**S. Richter**

Les champs **a_naiss** et **a_mort** contiennent respectivement l'année de naissance et l'année de la mort des compositeurs. La durée des oeuvres est fournie en minutes. Vous pouvez évidemment ajouter autant d'enregistrements d'oeuvres et de compositeurs que vous le voulez, mais ceux qui précèdent devraient suffire pour la suite de la démonstration.

Pour ce qui va suivre, nous supposerons donc que vous avez effectivement encodé les données des deux tables décrites ci-dessus. (Si vous éprouvez des difficultés à écrire le script nécessaire, nous en donnons un exemple dans les annexes de ces notes, à la page 271).

Le petit script ci-dessous est fourni à titre purement indicatif. Il s'agit d'un client SQL rudimentaire, qui vous permet de vous connecter à la base de données "musique" qui devrait à présent exister dans l'un de vos répertoires, d'y ouvrir un curseur et d'utiliser celui-ci pour effectuer des requêtes. Notez encore une fois que rien n'est transcrit sur le disque tant que la méthode **commit()** n'a pas été invoquée.

```
# Utilisation d'une petite base de données acceptant les requêtes SQL

import gadfly

connex = gadfly.gadfly("musique","\Python20\essais\gadfly")
cur = connex.cursor()
while 1:
    print "Veuillez entrer votre requête SQL (ou <Enter> pour terminer) :"
    requete = raw_input()
    if requete =="":
        break
    try:
        cur.execute(requete)        # tentative d'exécution de la requête SQL
    except:
        print '*** Requête incorrecte ***'
    else:
        print cur.pp()              # affichage du résultat de la requête
    print

choix = raw_input("Confirmez-vous l'enregistrement (o/n) ? ")
if choix[0] == "o" or choix[0] == "O":
    connex.commit()
else:
    connex.close()
```

Cette application très simple n'est évidemment qu'un exemple. Il faudrait y ajouter la possibilité de choisir la base de données ainsi que le répertoire de travail. Pour éviter que le script ne se "plante" lorsque l'utilisateur encode une requête incorrecte, nous avons utilisé ici le traitement des ***exceptions*** déjà décrit à la page 114.

16.2.4 La requête select

L'une des instructions les plus puissantes du langage SQL est l'instruction **select**, dont nous allons à présent explorer quelques fonctionnalités. Rappelons encore une fois que nous n'abordons ici qu'une très petite partie du sujet : la description détaillée de SQL peut occuper plusieurs livres.

Lancez donc le script ci-dessus, et analysez attentivement ce qui se passe lorsque vous proposez les requêtes suivantes :

select * from oeuvres

select * from oeuvres where comp = 'Mozart'

select comp, titre, duree from oeuvres order by comp

select titre, comp from oeuvres where comp='Beethoven' or comp='Mozart' order by comp

select count(*) from oeuvres

select sum(duree) from oeuvres

select avg(duree) from oeuvres

select sum(duree) from oeuvres where comp='Beethoven'

select * from oeuvres where duree >35 order by duree desc

Pour chacune de ces requêtes, tâchez d'exprimer le mieux possible ce qui se passe. Fondamentalement, vous activez sur la base de données des filtres de sélection et des tris.

Les requêtes suivantes sont plus élaborées, car elles concernent les deux tables à la fois.

select o.titre, c.nom, c.a_naiss from oeuvres o, compositeurs c where o.comp = c.comp

select comp from oeuvres intersect select comp from compositeurs

select comp from oeuvres except select comp from compositeurs

select comp from compositeurs except select comp from oeuvres

select distinct comp from oeuvres union select comp from compositeurs

Il ne nous est pas possible de développer davantage le langage de requêtes et la gestion des bases de données en général dans le cadre restreint de ces notes. Si le sujet vous intéresse, sachez que Python peut dialoguer avec de nombreux systèmes serveurs de bases de données, tels par exemple *PostgreSQL* ou *MySQL*. Après avoir installé les modules correspondants, il vous suffit de quelques instructions similaires à celles que nous avons décrites dans ces pages pour amorcer le dialogue.

Veuillez donc à ce sujet consulter les bons ouvrages de référence, comme par exemple "*Python : How to program*" de Deitel & coll. (voir bibliographie, page 8), ou encore les sites Web consacrés aux extensions de Python.

Chapitre 17 : Communications à travers un réseau

Le développement extraordinaire de l'internet a amplement démontré que les ordinateurs peuvent être des outils de communication très efficaces. Dans ce petit chapitre, nous allons expérimenter la plus simple des techniques d'interconnexion de deux programmes Python, qui leur permette de s'échanger des informations par l'intermédiaire d'un réseau.

Pour ce qui va suivre, nous supposerons donc que vous collaborez avec un ou plusieurs de vos condisciples, et que vos postes de travail Python sont connectés à un réseau local dont les communications utilisent le protocole TCP/IP. Le système d'exploitation n'a pas d'importance : vous pouvez par exemple installer l'un des scripts Python décrits ci-après sur un poste de travail fonctionnant sous Linux, et le faire dialoguer avec un autre script mis en oeuvre sur un poste de travail confié aux bons soins d'un système d'exploitation différent, tel que MacOS ou Windows.

17.1 Les sockets

Le premier exercice qui va vous être proposé consistera à établir une communication entre deux machines seulement. L'une et l'autre pourront s'échanger des messages à tour de rôle, mais vous constaterez cependant que leurs configurations ne sont pas symétriques. Le script installé sur l'une de ces machines jouera en effet le rôle d'un logiciel ***serveur***, alors que l'autre se comportera comme un logiciel ***client***.

Le logiciel serveur fonctionne en continu, sur une machine dont l'identité est bien définie sur le réseau grâce à une ***adresse IP*** spécifique[61]. Il guette en permanence l'arrivée de requêtes expédiées par les clients potentiels en direction de cette adresse, par l'intermédiaire d'un ***port de communication*** bien déterminé. Pour ce faire, le script correspondant doit mettre en oeuvre un objet logiciel associé à ce port, que l'on appelle un ***socket***.

Au départ d'une autre machine, le logiciel client tente d'établir la connexion en émettant une requête appropriée. Cette requête est un message qui est confié au réseau, un peu comme on confie une lettre à la Poste. Le réseau pourrait en effet acheminer la requête vers n'importe quelle autre machine, mais une seule est visée : pour que la destination visée puisse être atteinte, la requête contient dans son en-tête l'indication de l'adresse IP et du port de communication destinataires.

Lorsque la connexion est établie avec le serveur, le client lui assigne lui-même l'un de ses propres ports de communication. A partir de ce moment, on peut considérer qu'un canal privilégié relie les deux machines, comme si on les avait connectées l'une à l'autre par l'intermédiaire d'un fil (les deux ports de communication respectifs jouant le rôle des deux extrémités de ce fil). L'échange d'informations proprement dit peut commencer.

Pour pouvoir utiliser les ports de communication réseau, les programmes font appel à un ensemble de procédures et de fonctions du système d'exploitation par l'intermédiaire d'objets interfaces que l'on appelle des ***sockets***. Ceux-ci peuvent mettre en oeuvre deux techniques de communication différentes et complémentaires : celle des paquets (que l'on appelle aussi des datagrammes), très largement utilisée sur l'internet, et celle de la connexion continue, ou ***stream socket***, qui est un peu plus simple.

61 Une machine particulière peut également être désignée par un nom plus explicite, mais à la condition qu'un mécanisme ait été mis en place sur le réseau (DNS) pour traduire automatiquement ce nom en adresse IP. Veuillez consulter votre cours sur les systèmes d'exploitation et les réseaux pour en savoir davantage.

17.2 Construction d'un serveur élémentaire

Pour nos premières expériences, nous allons utiliser la technique des ***stream sockets***.
Celle-ci est en effet parfaitement appropriée lorsqu'il s'agit de faire communiquer des ordinateurs interconnectés par l'intermédiaire d'un réseau local. C'est une technique particulièrement aisée à mettre en oeuvre, et elle permet un débit élevé pour l'échange de données.

L'autre technologie (celle des paquets) serait préférable pour les communications expédiées via l'internet, en raison de sa plus grande fiabilité (les mêmes paquets peuvent atteindre leur destination par différents chemins, être émis ou ré-émis en plusieurs exemplaires si cela se révèle nécessaire pour corriger les erreurs de transmission), mais sa mise en oeuvre est un peu plus complexe. Nous la réserverons donc pour plus tard.

Le script ci-dessous met en place un serveur capable de communiquer avec un seul client. Nous verrons un peu plus loin ce qu'il faut lui ajouter afin qu'il puisse prendre en charge en parallèle les connexions de plusieurs clients.

```
# Définition d'un serveur réseau rudimentaire
# Ce serveur attend la connexion d'un client, pour entamer un dialogue avec lui

import socket, sys

HOST = '192.168.14.152'
PORT = 50000

# 1) création du socket :
mySocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# 2) liaison du socket à une adresse précise :
try:
    mySocket.bind((HOST, PORT))
except socket.error:
    print "La liaison du socket à l'adresse choisie a échoué."
    sys.exit

while 1:
    # 3) Attente de la requête de connexion d'un client :
    print "Serveur prêt, en attente de requêtes ..."
    mySocket.listen(5)

    # 4) Etablissement de la connexion :
    connexion, adresse = mySocket.accept()
    print "Client connecté, adresse IP %s, port %s" % (adresse[0], adresse[1])

    # 5) Dialogue avec le client :
    connexion.send("Vous êtes connecté au serveur Marcel. Envoyez vos messages.")
    msgClient = connexion.recv(1024)
    while 1:
        print "C>", msgClient
        if msgClient.upper() == "FIN" or msgClient =="":
            break
        msgServeur = raw_input("S> ")
        connexion.send(msgServeur)
        msgClient = connexion.recv(1024)

    # 6) Fermeture de la connexion :
    connexion.send("Au revoir !")
    print "Connexion interrompue."
    connexion.close()

    ch = raw_input("<R>ecommencer <T>erminer ? ")
    if ch.upper() =='T':
        break
```

Commentaires :

- Ligne 4 : Le module **socket** contient toutes les fonctions et les classes nécessaires pour construire des programmes communiquants. Comme nous allons le voir dans les lignes suivantes, l'établissement de la communication comporte six étapes.
- Lignes 6 & 7 : Ces deux variables définissent l'identité du serveur, telle qu'on l'intégrera au socket. HOST doit contenir une chaîne de caractères indiquant l'adresse IP du serveur sous la forme décimale habituelle, ou encore le nom DNS de ce même serveur (mais à la condition qu'un mécanisme de résolution des noms ait été mis en place sur le réseau). PORT doit contenir un entier, à savoir le numéro d'un port qui ne soit pas déjà utilisé pour un autre usage, et de préférence une valeur supérieure à 1024 (Cfr. votre cours sur les services réseau).
- Lignes 9 & 10 : Première étape du mécanisme d'interconnexion. On instancie un objet de la classe **socket()**, en précisant deux options qui indiquent le type d'adresses choisi (nous utiliserons des adresses de type "internet") ainsi que la technologie de transmission (datagrammes ou connexion continue (*stream)* : nous avons décidé d'utiliser cette dernière).
- Lignes 12 à 17 : Seconde étape. On tente d'établir la liaison entre le socket et le port de communication. Si cette liaison ne peut être établie (port de communication occupé, par exemple, ou nom de machine incorrect), le programme se termine sur un message d'erreur.
  Remarque : la méthode **bind()** du socket attend un argument du type tuple, raison pour laquelle nous devons enfermer nos deux variables dans une double paire de parenthèses.
- Ligne 19 : Notre programme serveur étant destiné à fonctionner en permanence dans l'attente des requêtes de clients potentiels, nous le lançons dans une boucle sans fin.
- Lignes 20 à 22 : Troisième étape. Le socket étant relié à un port de communication, il peut à présent se préparer à recevoir les requêtes envoyées par les clients. C'est le rôle de la méthode **listen()**. L'argument qu'on lui transmet indique le nombre maximum de connexions à accepter en parallèle. Nous verrons plus loin comment gérer celles-ci.
- Lignes 24 à 26 : Quatrième étape. Lorsqu'on fait appel à sa méthode **accept()**, le socket attend indéfiniment qu'une requête se présente. Le script est donc interrompu à cet endroit, un peu comme il le serait si nous faisions appel à une fonction **input()** pour attendre une entrée clavier. Si une requête est réceptionnée, la méthode **accept()** retourne un tuple de deux éléments : le premier est la référence d'un nouvel objet de la classe **socket()**[62], qui sera la véritable interface de communication entre le client et le serveur, et le second un autre tuple contenant les coordonnées de ce client (son adresse IP et le n° de port qu'il utilise lui-même).
- Lignes 28 à 30 : Cinquième étape. La communication proprement dite est établie. Les méthodes **send()** et **recv()** du socket servent évidemment à l'émission et à la réception des messages, qui doivent être de simples chaînes de caractères.
  Remarques : la méthode **send()** renvoie le nombre d'octets expédiés. L'appel de la méthode **recv()** doit comporter un argument entier indiquant le nombre maximum d'octets à réceptionner en une fois (Les octets surnuméraires sont mis en attente dans un tampon. Ils sont transmis lorsque la même méthode **recv()** est appelée à nouveau).

62 Nous verrons plus loin l'utilité de créer ainsi un nouvel objet socket pour prendre en charge la communication, plutôt que d'utiliser celui qui a déjà créé à la ligne 10. En bref, si nous voulons que notre serveur puisse prendre en charge simultanément les connexions de plusieurs clients, il nous faudra disposer d'un socket distinct pour chacun d'eux, indépendamment du premier que l'on laissera fonctionner en permanence pour réceptionner les requêtes qui continuent à arriver en provenance de nouveaux clients.

- Lignes 31 à 37 : Cette nouvelle boucle sans fin maintient le dialogue jusqu'à ce que le client décide d'envoyer le mot "fin" ou une simple chaîne vide. Les écrans des deux machines afficheront chacun l'évolution de ce dialogue.
- Lignes 39 à 42 : Sixième étape. Fermeture de la connexion.

17.3 Construction d'un client rudimentaire

Le script ci-dessous définit un logiciel client complémentaire du serveur décrit dans les pages précédentes. On notera sa grande simplicité.

```
1.  # Définition d'un client réseau rudimentaire
2.  # Ce client dialogue avec un serveur ad hoc
3.
4.  import socket, sys
5.
6.  HOST = '192.168.14.152'
7.  PORT = 50000
8.
9.  # 1) création du socket :
10. mySocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
11.
12. # 2) envoi d'une requête de connexion au serveur :
13. try:
14.     mySocket.connect((HOST, PORT))
15. except socket.error:
16.     print "La connexion a échoué."
17.     sys.exit()
18. print "Connexion établie avec le serveur."
19.
20. # 3) Dialogue avec le serveur :
21. msgServeur = mySocket.recv(1024)
22.
23. while 1:
24.     if msgServeur.upper() == "FIN" or msgServeur =="":
25.         break
26.     print "S>", msgServeur
27.     msgClient = raw_input("C> ")
28.     mySocket.send(msgClient)
29.     msgServeur = mySocket.recv(1024)
30.
31. # 4) Fermeture de la connexion :
32. print "Connexion interrompue."
33. mySocket.close()
```

Commentaires :

- Le début du script est similaire à celui du serveur. L'adresse IP et le port de communication doivent être ceux du serveur.
- Lignes 12 à 18 : On ne crée cette fois qu'un seul objet socket, dont on utilise la méthode **connect()** pour envoyer la requête de connexion.
- Lignes 20 à 33 : Une fois la connexion établie, on peut dialoguer avec le serveur en utilisant les méthodes **send()** et **recv()** déjà décrites plus haut pour celui-ci.

Annexes

0.1 Solutions de quelques exercices

Exercice 5 :

```
>>> c = 0
>>> while c < 20:
...     c = c +1
...     print c, "x 7 =", c*7
```

ou encore :

```
>>> c = 1
>>> while c <= 20:
...     print c, "x 7 =", c*7
...     c = c +1
```

Exercice 6 :

```
>>> s = 1
>>> while s <= 16384:
...     print s, "euro(s) =", s *1.65, "dollar(s)"
...     s = s *2
```

Exercice 7 :

```
>>> a, c = 1, 1
>>> while c < 13:
...     print a,
...     a, c = a *3, c+1
```

Exercice 9 :

```
# Le nombre de secondes est fourni au départ :
# (un grand nombre s'impose !)
nsd = 12345678912
# Nombre de secondes dans une journée :
nspj = 3600 * 24
# Nombre de secondes dans un an (soit 365 jours -
# on ne tiendra pas compte des années bissextiles) :
nspa = nspj * 365
# Nombre de secondes dans un mois (en admettant
# pour chaque mois une durée identique de 30 jours) :
nspm = nspj * 30
# Nombre d'années contenues dans la durée fournie :
na = nsd / nspa         # division <entière>
nsr = nsd % nspa        # n. de sec. restantes
# Nombre de mois restants :
nmo = nsr / nspm        # division <entière>
nsr = nsr % nspm        # n. de sec. restantes
# Nombre de jours restants :
nj = nsr / nspj         # division <entière>
nsr = nsr % nspj        # n. de sec. restantes
# Nombre d'heures restantes :
nh = nsr / 3600         # division <entière>
nsr = nsr % 3600        # n. de sec. restantes
```

```
# Nombre de minutes restantes :
nmi = nsr /60           # division <entière>
nsr = nsr % 60          # n. de sec. restantes

print "Nombre de secondes à convertir :", nsd
print "Cette durée correspond à", na, "années de 365 jours, plus"
print nmo, "mois de 30 jours,",
print nj, "jours,",
print nh, "heures,",
print nmi, "minutes et",
print nsr, "secondes."
```

Exercice 13 :

```
# Conversion degrés -> radians
# Rappel : un angle de 1 radian est un angle qui correspond à une portion
# de circonférence de longueur égale à celle du rayon.
# Puisque la circonférence vaut 2 pi R, un angle de 1 radian correspond
# à 360° / 2 pi , ou encore à 180° / pi

# Angle fourni au départ en degrés, minutes, secondes :
deg, min, sec  = 32, 13, 49

# Conversion des secondes en une fraction de minute :
# (le point décimal force la conversion du résultat en un nombre réel)
fm = sec/60.
# Conversion des minutes en une fraction de degré :
fd = (min + fm)/60
# Valeur de l'angle en degrés "décimalisés" :
ang = deg + fd
# Valeur de pi :
pi = 3.14159265359
# Valeur d'un radian en degrés :
rad = 180 / pi
# Conversion de l'angle en radians :
arad = ang / rad
# Affichage :
print deg, "°", min, "'", sec, '" =', arad, "radian(s)"
```

Exercice 15 :

```
# Conversion °Fahrenheit <-> °Celsius

# A) Température fournie en °C :
tempC = 25
# Conversion en °Fahrenheit :
tempF = tempC * 1.8 + 32
# Affichage :
print tempC, "°C =", tempF, "°F"

# B) Température fournie en °F :
tempF = 25
# Conversion en °Celsius :
tempC = (tempF - 32) / 1.8
# Affichage :
print tempF, "°F =", tempC, "°C"
```

Exercice 17 (problème de l'échiquier) :

```
>>> a, b = 1, 1                  # variante :  a, b = 1., 1
>>> while b<65:
...     print b, a
...     a,b = a*2, b+1
...
```

Exercice 18 :

```
# Recherche d'un caractère particulier dans une chaîne

# Chaîne fournie au départ :
ch = "Monty python flying circus"
# Caractère à rechercher :
cr = "e"
# Recherche proprement dite :
lc = len(ch)    # nombre de caractères à tester
i = 0           # indice du caractère en cours d'examen
t = 0           # "drapeau" à lever si le caractère recherché est présent
while i < lc:
    if ch[i] == cr:
        t = 1
    i = i + 1
# Affichage :
print "Le caractère", cr,
if t == 1:
    print "est présent",
else:
    print "est inrouvable",
print "dans la chaîne", ch
```

Exercice 20 :

```
# Insertion d'un caractère d'espacement dans une chaîne

# Chaîne fournie au départ :
ch = "Gaston"
# Caractère à insérer :
cr = "*"
# Le nombre de caractères à insérer est inférieur d'une unité au
# nombre de caractères de la chaîne. On traitera donc celle-ci à
# partir de son second caractère (en omettant le premier).
lc = len(ch)    # nombre de caractères total
i = 1           # indice du premier caractère à examiner (le second, en fait)
nch = ch[0]     # nouvelle chaîne à construire (contient déjà le premier car.)
while i < lc:
    nch = nch + cr + ch[i]
    i = i + 1
# Affichage :
print nch
```

Exercice 21 :

```
# Inversion d'une chaîne de caractères

# Chaîne fournie au départ :
ch = "zorglub"
lc = len(ch)    # nombre de caractères total
i = lc - 1      # le traitement commencera à partir du dernier caractère
```

```
nch = ""        # nouvelle chaîne à construire (vide au départ)
while i >= 0:
    nch = nch + ch[i]
    i = i - 1
# Affichage :
print nch
```

Exercice 23 :

```
# Combinaison de deux listes en une seule

# Listes fournies au départ :
t1 = [31,28,31,30,31,30,31,31,30,31,30,31]
t2 = ['Janvier','Février','Mars','Avril','Mai','Juin',
      'Juillet','Août','Septembre','Octobre','Novembre','Décembre']
# Nouvelle liste à construire (vide au départ) :
t3 = []
# Boucle de traitement :
i = 0
while i < len(t1):
    t3.append(t2[i])
    t3.append(t1[i])
    i = i + 1

# Affichage :
print t3
```

Exercice 24 :

```
# Affichage des éléments d'une liste

# Liste fournie au départ :
t2 = ['Janvier','Février','Mars','Avril','Mai','Juin',
      'Juillet','Août','Septembre','Octobre','Novembre','Décembre']
# Affichage :
i = 0
while i < len(t2):
    print t2[i],
    i = i + 1
```

Exercice 25 :

```
# Recherche du plus grand élément d'une liste

# Liste fournie au départ :
tt = [32, 5, 12, 8, 3, 75, 2, 15]
# Au fur et à mesure du traitement de la liste, on mémorisera dans
# la variable ci-dessous la valeur du plus grand élément déjà trouvé :
max = 0
# Examen de tous les éléments :
i = 0
while i < len(tt):
    if tt[i] > max:
        max = tt[i]         # mémorisation d'un nouveau maximum
    i = i + 1
# Affichage :
print "Le plus grand élément de cette liste a la valeur", max
```

Exercice 26 :

```
# Séparation des nombres pairs et impairs

# Liste fournie au départ :
tt = [32, 5, 12, 8, 3, 75, 2, 15]
pairs = []
impairs = []
# Examen de tous les éléments :
i = 0
while i < len(tt):
    if tt[i] % 2 == 0:
        pairs.append(tt[i])
    else:
        impairs.append(tt[i])
    i = i + 1
# Affichage :
print "Nombres pairs :", pairs
print "Nombres impairs :", impairs
```

Exercice 28 :

```
# Conversion de miles/heure en km/h et m/s

print "Veuillez entrer le nombre de miles parcourus en une heure : ",
ch = raw_input()            # en général préférable à input()
mph = float(ch)             # conversion de la chaîne entrée en nombre réel
mps = mph * 1609 / 3600     # conversion en mètres par seconde
kmph = mph * 1.609          # conversion en km/h
# affichage :
print mph, "miles/heure =", kmph, "km/h, ou encore", mps, "m/s"
```

Exercice 29 :

```
# Périmètre et Aire d'un triangle quelconque

from math import sqrt

print "Veuillez entrer le côté a : "
a = float(raw_input())
print "Veuillez entrer le côté b : "
b = float(raw_input())
print "Veuillez entrer le côté c : "
c = float(raw_input())
d = (a + b + c)/2                # demi-périmètre
s = sqrt(d*(d-a)*(d-b)*(d-c))    # aire (suivant formule)

print "Longueur des côtés =", a, b, c
print "Périmètre =", d*2, "Aire =", s
```

Exercice 31 :

```
# Entrée d'éléments dans une liste

tt = []              # Liste à compléter (vide au départ)
ch = "start"         # valeur quelconque (mais non nulle)
while ch != "":
    print "Veuillez entrer une valeur : "
    ch = raw_input()
    if ch != "":
        tt.append(float(ch))        # variante : tt.append(ch)

# affichage de la liste :
print tt
```

Exercice 34 :

```
# Traitement de nombres entiers compris entre deux limites

print "Veuillez entrer la limite inférieure :",
a = input()
print "Veuillez entrer la limite supérieure :",
b = input()
s = 0                    # somme recherchée (nulle au départ)
# Parcours de la série des nombres compris entre a et b :
n = a                    # nombre en cours de traitement
while n <= b:
    if n % 3 ==0 and n % 5 ==0:      # variante : 'or' au lieu de 'and'
        s = s + n
    n = n + 1

print "La somme recherchée vaut", s
```

Exercice 35 :

```
# Années bissextiles

print "Veuillez entrer l'année à tester :",
a = input()

if a % 4 != 0:
    # a n'est pas divisible par 4 -> année non bissextile
    bs = 0
else:
    if a % 400 ==0:
        # a divisible par 400 -> année bissextile
        bs = 1
    elif a % 100 ==0:
        # a divisible par 100 -> année non bissextile
        bs = 0
    else:
        # autres cas ou a est divisible par 4 -> année bissextile
        bs = 1
if bs ==1:
    ch = "est"
else:
    ch = "n'est pas"
print "L'année", a, ch, "bissextile."
```

Exercice 37 :

```
# Calculs de triangles

from sys import exit      # module contenant des fonctions système

print """
Veuillez entrer les longueurs des 3 côtés
(en séparant ces valeurs à l'aide de virgules) :"""
a, b, c = input()
# Il n'est possible de construire un triangle que si chaque côté
# a une longueur inférieure à la somme des deux autres :
if a < (b+c) and b < (a+c) and c < (a+b) :
    print "Ces trois longueurs déterminent bien un triangle."
else:
    print "Il est impossible de construire un tel triangle !"
    exit()          # ainsi l'on n'ira pas plus loin.

f = 0
if a == b and b == c :
    print "Ce triangle est équilatéral."
    f = 1
elif a == b or b == c or c == a :
    print "Ce triangle est isocèle."
    f = 1
if a*a + b*b == c*c or b*b + c*c == a*a or c*c + a*a == b*b :
    print "Ce triangle est rectangle."
    f = 1
if f == 0 :
    print "Ce triangle est quelconque."
```

Exercice 41 :

```
# Notes de travaux scolaires

notes = []           # liste à construire
n = 2                # valeur positive quelconque pour initier la boucle
while n >= 0 :
    print "Entrez la note suivante, s.v.p. : ",
    n = float(raw_input())      # conversion de l'entrée en un nombre réel
    if n < 0 :
        print "OK. Terminé."
    else:
        notes.append(n)         # ajout d'une note à la liste
        # Calculs divers sur les notes déjà entrées :
        # valeurs minimale et maximale + total de toutes les notes.
        min = 500               # valeur supérieure à toute note
        max, tot, i = 0, 0, 0
        nn = len(notes)         # nombre de notes déjà entrées
        while i < nn:
            if notes[i] > max:
                max = notes[i]
            if notes[i] < min:
                min = notes[i]
            tot = tot + notes[i]
            moy = tot/nn
            i = i + 1
        print nn, "notes entrées. Max =", max, "Min =", min, "Moy =", moy
```

Exercice 45 :

```
from math import pi

def surfCercle(r):
    "Surface d'un cercle de rayon r"
    return pi * r**2

# test :
print surfCercle(2.5)
```

Exercice 46 :

```
def volBoite(x1, x2, x3):
    "Volume d'une boîte parallélipipédique"
    return x1 * x2 * x3

# test :
print volBoite(5.2, 7.7, 3.3)
```

Exercice 47 :

```
def maximum(n1, n2, n3):
    "Renvoie le plus grand de trois nombres"
    if n1 >= n2 and n1 >= n3:
        return n1
    elif n2 >= n1 and n2 >= n3:
        return n2
    else:
        return n3

# test :
print maximum(4.5, 5.7, 3.9)
```

Exercice 51 :

```
def compteCar(ca, ch):
    "Renvoie le nombre de caractères ca trouvés dans la chaîne ch"
    i, tot = 0, 0
    while i < len(ch):
        if ch[i] == ca:
            tot = tot + 1
        i = i + 1
    return tot

# test :
print compteCar("e","Cette chaîne est un exemple")
```

Exercice 52 :

```
def indexMax(tt):
    "renvoie l'indice du plus grand élément de la liste tt"
    i, max = 0, 0
    while i < len(tt):
        if tt[i] > max :
            max, imax = tt[i], i
        i = i + 1
    return imax

# test :
serie = [5, 8, 2, 1, 9, 3, 6, 4]
print indexMax(serie)
```

Exercice 53 :

```
def nomMois(n):
    "renvoie le nom du n-ième mois de l'année"
    mois = ['Janvier,', 'Février', 'Mars', 'Avril', 'Mai', 'Juin', 'Juillet',
            'Août', 'Septembre', 'Octobre', 'Novembre', 'Décembre']
    return mois[n -1]       # les indices sont numérotés à partir de zéro

# test :
print nomMois(4)
```

Exercice 56 :

```
def volBoite(x1 =10, x2 =10, x3 =10):
    "Volume d'une boîte parallélipipédique"
    return x1 * x2 * x3

# test :
print volBoite()
print volBoite(5.2)
print volBoite(5.2, 3)
```

Exercice 57 :

```
def volBoite(x1 =-1, x2 =-1, x3 =-1):
    "Volume d'une boîte parallélipipédique"
    if x1 == -1 :
        return x1           # aucun argument n'a été fourni
    elif x2 == -1 :
        return x1**3        # un seul argument -> boîte cubique
    elif x3 == -1 :
        return x1*x1*x2     # deux arguments -> boîte prismatique
    else :
        return x1*x2*x3

# test :
print volBoite()
print volBoite(5.2)
print volBoite(5.2, 3)
print volBoite(5.2, 3, 7.4)
```

Exercice 58 :

```
def changeCar(ch, ca1, ca2, debut =0, fin =-1):
    "Remplace tous les caractères ca1 par des ca2 dans la chaîne ch"
    if fin == -1:
        fin = len(ch)
    nch, i = "", 0            # nch : nouvelle chaîne à construire
    while i < len(ch) :
        if i >= debut and i <= fin and ch[i] == ca1:
            nch = nch + ca2
        else :
            nch = nch + ch[i]
        i = i + 1
    return nch

# test :
print changeCar("Ceci est une toute petite phrase", " ", "*")
print changeCar("Ceci est une toute petite phrase", " ", "*", 8, 12)
print changeCar("Ceci est une toute petite phrase", " ", "*", 12)
```

Exercice 59 :

```
def eleMax(lst, debut =0, fin =-1):
    "renvoie le plus grand élément de la liste lst"
    if fin == -1:
        fin = len(lst)
    max, i = 0, 0
    while i < len(lst):
        if i >= debut and i <= fin and lst[i] > max:
            max = lst[i]
        i = i + 1
    return max

# test :
serie = [9, 3, 6, 1, 7, 5, 4, 8, 2]
print eleMax(serie)
print eleMax(serie, 2)
print eleMax(serie, 2, 5)
```

Exercice 66 :

```
from Tkinter import *

# Coordonnées X,Y des 5 anneaux :
coord = [[20,30], [120,30], [220, 30], [70,80], [170,80]]
# Couleurs des 5 anneaux :
coul = ["red", "yellow", "blue", "green", "black"]

base = Tk()
can = Canvas(base, width =335, height =200, bg ="white")
can.pack()
bou = Button(base, text ="Quitter", command =base.quit)
bou.pack(side = RIGHT)
# Dessin des 5 anneaux :
i = 0
while i < 5:
    x1, y1 = coord[i][0], coord[i][1]
    can.create_oval(x1, y1, x1+100, y1 +100, width =2, outline =coul[i])
    i = i +1
base.mainloop()
```

Variante :

```
from Tkinter import *

# Dessin des 5 anneaux :
def dessineCercle(i):
    x1, y1 = coord[i][0], coord[i][1]
    can.create_oval(x1, y1, x1+100, y1 +100, width =2, outline =coul[i])

def a1():
    dessineCercle(0)

def a2():
    dessineCercle(1)

def a3():
    dessineCercle(2)

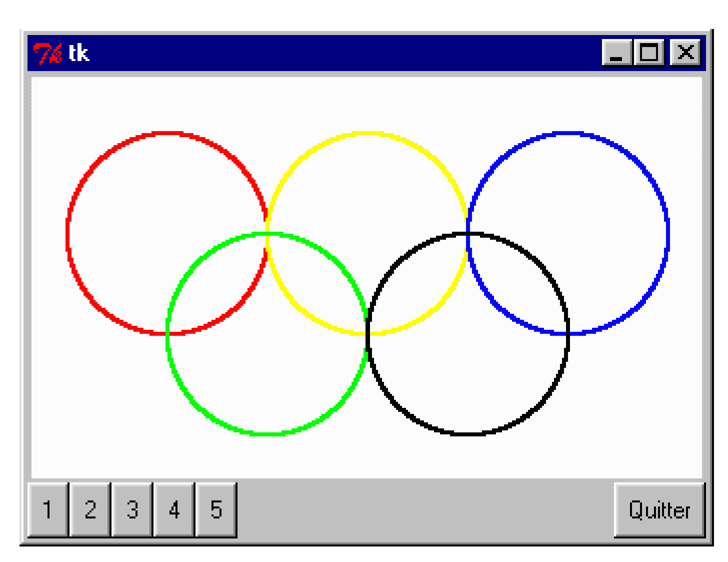
def a4():
    dessineCercle(3)

def a5():
    dessineCercle(4)

# Coordonnées X,Y des 5 anneaux :
coord = [[20,30], [120,30], [220, 30], [70,80], [170,80]]
# Couleurs des 5 anneaux :
coul = ["red", "yellow", "blue", "green", "black"]

base = Tk()
can = Canvas(base, width =335, height =200, bg ="white")
can.pack()
bou = Button(base, text ="Quitter", command =base.quit)
bou.pack(side = RIGHT)

# Installation des 5 boutons :
Button(base, text='1', command = a1).pack(side =LEFT)
Button(base, text='2', command = a2).pack(side =LEFT)
Button(base, text='3', command = a3).pack(side =LEFT)
Button(base, text='4', command = a4).pack(side =LEFT)
Button(base, text='5', command = a5).pack(side =LEFT)
base.mainloop()
```



Exercice 69 :

```
# Simulation du phénomène de gravitation universelle

from Tkinter import *
from math import sqrt

def distance(x1, y1, x2, y2):
    "distance séparant les points x1,y1 et x2,y2"
    d = sqrt((x2-x1)**2 + (y2-y1)**2)       # théorème de Pythagore
    return  d

def forceG(m1, m2, di):
    "force de gravitation s'exerçant entre m1 et m2 pour une distance di"
    return m1*m2*6.67e-11/di**2             # loi de Newton

def avance(n, gd, hb):
    "déplacement de l'astre n, de gauche à droite ou de haut en bas"
    global x, y, step
    # nouvelles coordonnées :
    x[n], y[n] = x[n] +gd, y[n] +hb
    # déplacement du dessin dans le canevas :
    can.coords(astre[n], x[n]-10, y[n]-10, x[n]+10, y[n]+10)
    # calcul de la nouvelle interdistance :
    di = distance(x[0], y[0], x[1], y[1])
    # conversion de la distance "écran" en distance "astronomique" :
    diA = di*1e9            # (1 pixel => 1 million de km)
    # calcul de la force de gravitation correspondante :
    f = forceG(m1, m2, diA)
    # affichage des nouvelles valeurs de distance et force :
    valDis.configure(text="Distance = " +str(diA) +" m")
    valFor.configure(text="Force = " +str(f) +" N")
    # adaptation du "pas" de déplacement en fonction de la distance :
    step = di/10

def gauche1():
    avance(0, -step, 0)

def droite1():
    avance(0, step, 0)

def haut1():
    avance(0, 0, -step)

def bas1():
    avance(0, 0, step)

def gauche2():
    avance(1, -step, 0)

def droite2():
    avance (1, step, 0)

def haut2():
    avance(1, 0, -step)

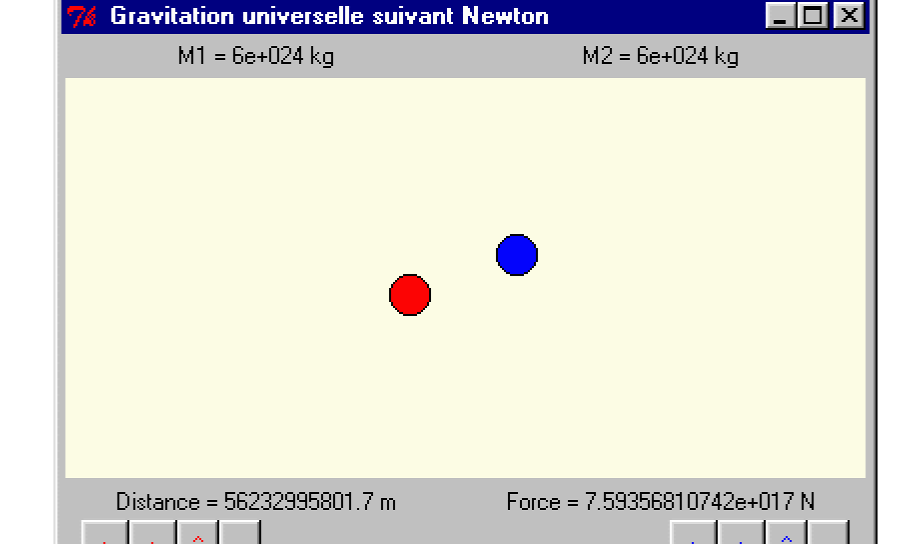
def bas2():
    avance(1, 0, step)

# Masses des deux astres :
m1 = 6e24          # (valeur de la masse de la terre, en kg)
m2 = 6e24          #
```

```
astre = [0]*2        # liste servant à mémoriser les références des dessins
x =[50., 350.]       # liste des coord. X de chaque astre (à l'écran)
y =[100., 100.]      # liste des coord. Y de chaque astre
step =10             # "pas" de déplacement initial

# Construction de la fenêtre :
fen = Tk()
fen.title(' Gravitation universelle suivant Newton')
# Libellés :
valM1 = Label(fen, text="M1 = " +str(m1) +" kg")
valM1.grid(row =1, column =0)
valM2 = Label(fen, text="M2 = " +str(m2) +" kg")
valM2.grid(row =1, column =1)
valDis = Label(fen, text="Distance")
valDis.grid(row =3, column =0)
valFor = Label(fen, text="Force")
valFor.grid(row =3, column =1)
# Canevas avec le dessin des 2 astres:
can = Canvas(fen, bg ="light yellow", width =400, height =200)
can.grid(row =2, column =0, columnspan =2)
astre[0] = can.create_oval(x[0]-10, y[0]-10, x[0]+10, y[0]+10,
                           fill ="red", width =1)
astre[1] = can.create_oval(x[1]-10, y[1]-10, x[1]+10, y[1]+10,
                           fill ="blue", width =1)
# 2 groupes de 4 boutons, chacun installé dans un cadre (frame) :
fra1 = Frame(fen)
fra1.grid(row =4, column =0, sticky =W, padx =10)
Button(fra1, text="<-", fg ='red',command =gauche1).pack(side =LEFT)
Button(fra1, text="->", fg ='red', command =droite1).pack(side =LEFT)
Button(fra1, text="^", fg ='red', command =haut1).pack(side =LEFT)
Button(fra1, text="v", fg ='red', command =bas1).pack(side =LEFT)
fra2 = Frame(fen)
fra2.grid(row =4, column =1, sticky =E, padx =10)
Button(fra2, text="<-", fg ='blue', command =gauche2).pack(side =LEFT)
Button(fra2, text="->", fg ='blue', command =droite2).pack(side =LEFT)
Button(fra2, text="^", fg ='blue', command =haut2).pack(side =LEFT)
Button(fra2, text="v", fg ='blue', command =bas2).pack(side =LEFT)

fen.mainloop()
```



Exercice 73 :

```
# Conversions de températures Fahrenheit <=> Celsius

from Tkinter import *

def convFar(event):
    "valeur de cette température, exprimée en degrés Fahrenheit"
    tF = eval(champTC.get())
    varTF.set(str(tF*1.8 +32))

def convCel(event):
    "valeur de cette température, exprimée en degrés Celsius"
    tC = eval(champTF.get())
    varTC.set(str((tC-32)/1.8))

fen = Tk()
fen.title('Fahrenheit/Celsius')

Label(fen, text='Temp. Celsius :').grid(row =0, column =0)
# "variable Tkinter" associée au champ d'entrée. Cet "objet-variable"
# assure l'interface entre TCL et Python (voir notes, page 165) :
varTC =StringVar()
champTC = Entry(fen, textvariable =varTC)
champTC.bind("<Return>", convFar)
champTC.grid(row =0, column =1)
# Initialisation du contenu de la variable Tkinter :
varTC.set("100.0")

Label(fen, text='Temp. Fahrenheit :').grid(row =1, column =0)
varTF =StringVar()
champTF = Entry(fen, textvariable =varTF)
champTF.bind("<Return>", convCel)
champTF.grid(row =1, column =1)
varTF.set("212.0")

fen.mainloop()
```

Exercice 75 à 77 :

```
# Cercles et courbes de Lissajous

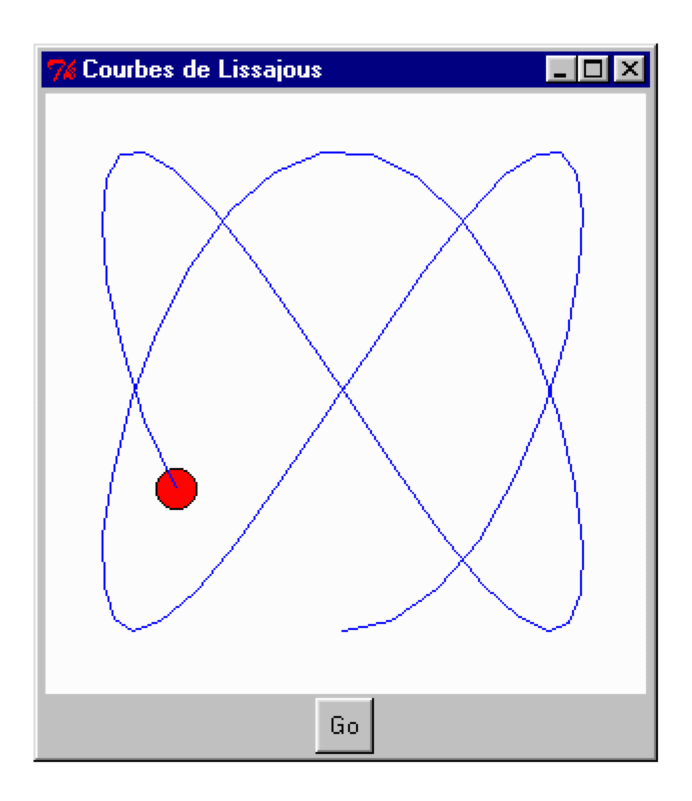
from Tkinter import *
from math import sin, cos

def move():
    global ang, x, y
    # on mémorise les coord. précédentes avant de calculer les nouvelles :
    xp, yp = x, y
    # rotation d'un angle de 0.1 radian :
    ang = ang +.1
    # sinus et cosinus de cet angle => coord. d'un point du cercle trigono.
    x, y = sin(ang), cos(ang)
    # Variante déterminant une courbe de Lissajous avec f1/f2 = 2/3 :
    # x, y = sin(2*ang), cos(3*ang)
```

```
    # mise à l'échelle (120 = rayon du cercle, (150,150) = centre du canevas)
    x, y = x*120 + 150, y*120 + 150
    can.coords(balle, x-10, y-10, x+10, y+10)
    can.create_line(xp, yp, x, y, fill ="blue")

ang, x, y = 0., 150., 270.
fen = Tk()
fen.title('Courbes de Lissajous')
can = Canvas(fen, width =300, height=300, bg="white")
can.pack()
balle = can.create_oval(x-10, y-10, x+10, y+10, fill='red')
Button(fen, text='Go', command =move).pack()

fen.mainloop()
```



Exercice 84 :

```
# Chutes et rebonds

from Tkinter import *

def move():
    global x, y, v, dx, dv, flag
    xp, yp = x, y            # mémorisation des coord. précédentes
    # déplacement horizontal :
    if x > 385 or x < 15 :   # rebond sur les parois latérales :
        dx = -dx             # on inverse le déplacement
    x = x + dx
    # variation de la vitesse verticale (toujours vers le bas):
    v = v + dv
```

```
    # déplacement vertical (proportionnel à la vitesse)
    y = y + v
    if y > 240:                # niveau du sol à 240 pixels :
        y = 240                #  défense d'aller + loin !
        v = -v                 # rebond : la vitesse s'inverse
    # on repositionne la balle :
    can.coords(balle, x-10, y-10, x+10, y+10)
    # on trace un bout de trajectoire :
    can.create_line(xp, yp, x, y, fill ='light grey')
    # ... et on remet ça jusqu'à plus soif :
    if flag > 0:
        fen.after(50,move)

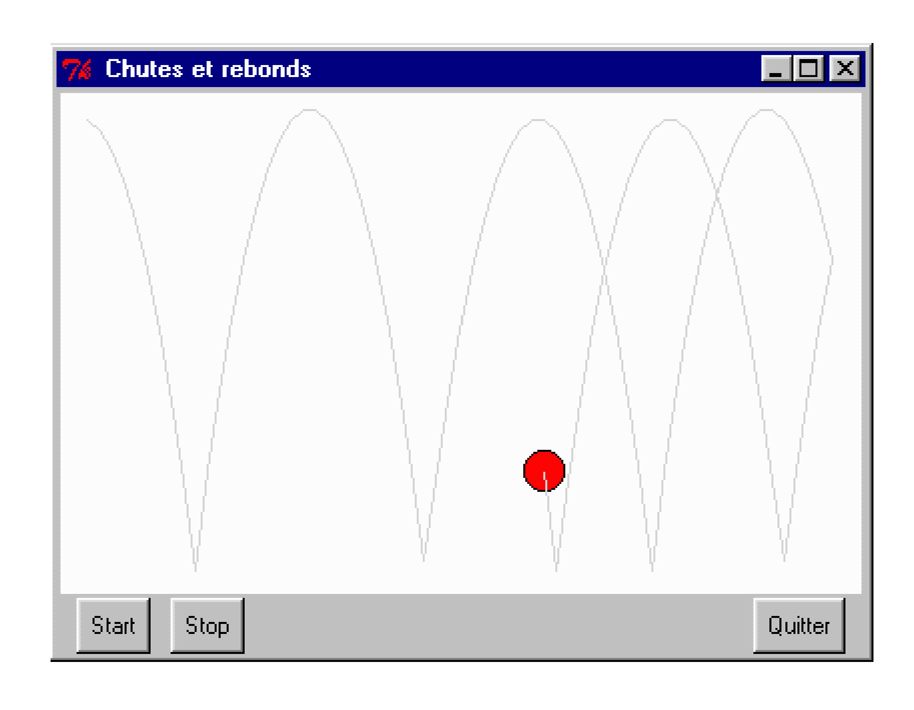
def start():
    global flag
    flag = flag +1
    if flag == 1:
        move()

def stop():
    global flag
    flag =0

# initialisation des coordonnées, des vitesses et du témoin d'animation :
x, y, v, dx, dv, flag  = 15, 15, 0, 6, 5, 0

fen = Tk()
fen.title(' Chutes et rebonds')
can = Canvas(fen, width =400, height=250, bg="white")
can.pack()
balle = can.create_oval(x-10, y-10, x+10, y+10, fill='red')
Button(fen, text='Start', command =start).pack(side =LEFT, padx =10)
Button(fen, text='Stop', command =stop).pack(side =LEFT)
Button(fen, text='Quitter', command =fen.quit).pack(side =RIGHT, padx =10)

fen.mainloop()
```



Exercice 92 (Editeur simple, pour lire et écrire dans un fichier 'texte') :

```
def sansDC(ch):
    "cette fonction renvoie la chaîne ch amputée de son dernier caractère"
    nouv = ""
    i, j = 0, len(ch) -1
    while i < j:
        nouv = nouv + ch[i]
        i = i + 1
    return nouv

def ecrireDansFichier():
    of = open(nomF, 'a')
    while 1:
        ligne = raw_input("entrez une ligne de texte (ou <Enter>) : ")
        if ligne == '':
            break
        else:
            of.write(ligne + '\n')
    of.close()

def lireDansFichier():
    of = open(nomF, 'r')
    while 1:
        ligne = of.readline()
        if ligne == "":
            break
        # afficher en omettant le dernier caractère (= fin de ligne) :
        print sansDC(ligne)
    of.close()

nomF = raw_input('Nom du fichier à traiter : ')
choix = raw_input('Entrez "e" pour écrire, "c" pour consulter les données : ')

if choix =='e':
    ecrireDansFichier()
else:
    lireDansFichier()
```

Exercice 94 (génération des tables de multiplication de 2 à 30) :

```
def tableMulti(n):
    # Fonction générant la table de multiplication par n (20 termes)
    # La table sera retournée sous forme d'une chaîne de caractères :
    i, ch = 0, ""
    while i < 20:
        i = i + 1
        ch = ch + str(i * n) + " "
    return ch

NomF = raw_input("Nom du fichier à créer : ")
fichier = open(NomF, 'w')

# Génération des tables de 2 à 30 :
table = 2
while table < 31:
    fichier.write(tableMulti(table) + '\n')
    table = table + 1
fichier.close()
```

Exercice 95 (triplement des espaces dans un fichier texte) :

```
# Ce script montre également comment modifier le contenu d'un fichier
# en le transférant d'abord tout entier dans une liste, puis en
# réenregistrant celle-ci après modifications

def triplerEspaces(ch):
    "fonction qui triple les espaces entre mots dans la chaîne ch"
    i, nouv = 0, ""
    while i < len(ch):
        if ch[i] == " ":
            nouv = nouv + "   "
        else:
            nouv = nouv + ch[i]
        i = i +1
    return nouv

NomF = raw_input("Nom du fichier : ")
fichier = open(NomF, 'r+')              # 'r+' = mode read/write
lignes = fichier.readlines()            # lire toutes les lignes

n=0
while n < len(lignes):
    lignes[n] = triplerEspaces(lignes[n])
    n =n+1

fichier.seek(0)                         # retour au début du fichier
fichier.writelines(lignes)              # réenregistrement
fichier.close()
```

Exercice 96 (mise en forme de données numériques) :

```
# Le fichier traité est un fichier texte dont chaque ligne contient un nombre
# réel (sans exposants et encodé sous la forme d'une chaîne de caractères)

def valArrondie(ch):
    "représentation arrondie du nombre présenté dans la chaîne ch"
    f = float(ch)       # conversion de la chaîne en un nombre réel
    e = int(f + .5)     # conversion en entier (On ajoute d'abord
                        # 0.5 au réel pour l'arrondir correctement)
    return str(e)       # reconversion en chaîne de caractères

fiSource = raw_input("Nom du fichier à traiter : ")
fiDest = raw_input("Nom du fichier destinataire : ")
fs = open(fiSource, 'r')
fd = open(fiDest, 'w')

while 1:
    ligne = fs.readline()       # lecture d'une ligne du fichier
    if ligne == "" or ligne == "\n":
        break
    ligne = valArrondie(ligne)
    fd.write(ligne +"\n")

fd.close()
fs.close()
```

Exercice 97 :

```
# Comparaison de deux fichiers, caractère par caractère :

fich1 = raw_input("Nom du premier fichier : ")
fich2 = raw_input("Nom du second fichier : ")
fi1 = open(fich1, 'r')
fi2 = open(fich2, 'r')

c, f = 0, 0                  # compteur de caractères et "drapeau"
while 1:
    c = c + 1
    car1 = fi1.read(1)       # lecture d'un caractère dans chacun
    car2 = fi2.read(1)       # des deux fichiers
    if car1 =="" or car2 =="":
        break
    if car1 != car2 :
        f = 1
        break                # différence trouvée

fi1.close()
fi2.close()

print "Ces 2 fichiers",
if f ==1:
    print "diffèrent à partir du caractère n°", c
else:
    print "sont identiques."
```

Exercice 98 :

```
# Combinaison de deux fichiers texte pour en faire un nouveau

fichA = raw_input("Nom du premier fichier : ")
fichB = raw_input("Nom du second fichier : ")
fichC = raw_input("Nom du fichier destinataire : ")
fiA = open(fichA, 'r')
fiB = open(fichB, 'r')
fiC = open(fichC, 'w')

while 1:
    ligneA = fiA.readline()
    ligneB = fiB.readline()
    if ligneA =="" and ligneB =="":
        break                # On est arrivé à la fin des 2 fichiers
    if ligneA != "":
        fiC.write(ligneA)
    if ligneB != "":
        fiC.write(ligneB)

fiA.close()
fiB.close()
fiC.close()
```

Exercice 99 :

```
# Enregistrer les coordonnées des membres d'un club

def encodage():
    "renvoie la liste des valeurs entrées, ou une liste vide"
    print "*** Veuillez entrer les données (ou <Enter> pour terminer) :"
    while 1:
        nom = raw_input("Nom : ")
        if nom == "":
            return []
        prenom = raw_input("Prénom : ")
        rueNum = raw_input("Adresse (N° et rue) : ")
        cPost = raw_input("Code postal : ")
        local = raw_input("Localité : ")
        tel = raw_input("N° de téléphone : ")
        print nom, prenom, rueNum, cPost, local, tel
        ver = raw_input("Entrez <Enter> si c'est correct, sinon <n> ")
        if ver == "":
            break
    return [nom, prenom, rueNum, cPost, local, tel]

def enregistrer(liste):
    "enregistre les données de la liste en les séparant par des <#>"
    i = 0
    while i < len(liste):
        of.write(liste[i] + "#")
        i = i + 1
    of.write("\n")              # caractère de fin de ligne

nomF = raw_input('Nom du fichier destinataire : ')
of = open(nomF, 'a')
while 1:
    tt = encodage()
    if tt == []:
        break
    enregistrer(tt)

of.close()
```

Exercice 100 :

```
# Ajouter des informations dans le fichier du club

def traduire(ch):
    "convertir une ligne du fichier source en liste de données"
    dn = ""                 # chaîne temporaire pour extraire les données
    tt = []                 # la liste à produire
    i = 0
    while i < len(ch):
        if ch[i] == "#":
            tt.append(dn)   # on ajoute la donnée à la liste, et
            dn =""          # on réinitialise la chaine temporaire
        else:
            dn = dn + ch[i]
        i = i + 1
    return tt

def encodage(tt):
    "renvoyer la liste tt, complétée avec la date de naissance et le sexe"
    print "*** Veuillez entrer les données (ou <Enter> pour terminer) :"
    # Affichage des données déjà présentes dans la liste :
    i = 0
    while i < len(tt):
        print tt[i],
        i = i +1
    print
    while 1:
        daNai = raw_input("Date de naissance : ")
        sexe = raw_input("Sexe (m ou f) : ")
        print daNai, sexe
        ver = raw_input("Entrez <Enter> si c'est correct, sinon <n> ")
        if ver == "":
            break
    tt.append(daNai)
    tt.append(sexe)
    return tt

def enregistrer(tt):
    "enregistrer les données de la liste tt en les séparant par des <#>"
    i = 0
    while i < len(tt):
        fd.write(tt[i] + "#")
        i = i + 1
    fd.write("\n")          # caractère de fin de ligne

fSource = raw_input('Nom du fichier source : ')
fDest = raw_input('Nom du fichier destinataire : ')
fs = open(fSource, 'r')
fd = open(fDest, 'w')
while 1:
    ligne = fs.readline()           # lire une ligne du fichier source
    if ligne =="" or ligne =="\n":
        break
    liste = traduire(ligne)         # la convertir en une liste
    liste = encodage(liste)         # y ajouter les données supplémentaires
    enregistrer(liste)              # sauvegarder dans fichier dest.

fd.close()
fs.close()
```

Exercice 101 :

```
# Recherche de lignes particulières dans un fichier texte :

def chercheCP(ch):
    "recherche dans ch la portion de chaîne contenant le code postal"
    i, f, ns = 0, 0, 0          # ns est un compteur de codes #
    cc = ""                     # chaîne à construire
    while i < len(ch):
        if ch[i] =="#":
            ns = ns +1
            if ns ==3:          # le CP se trouve après le 3e code #
                f = 1           # variable "drapeau" (flag)
            elif ns ==4:        # inutile de lire après le 4e code #
                break
        elif f ==1:             # le caractère lu fait partie du
            cc = cc + ch[i]     # CP recherché -> on mémorise
        i = i +1
    return cc

nomF = raw_input("Nom du fichier à traiter : ")
codeP = raw_input("Code postal à rechercher : ")
fi = open(nomF, 'r')
while 1:
    ligne = fi.readline()
    if ligne =="":
        break
    if chercheCP(ligne) == codeP:
        print ligne
fi.close()
```

Exercice 105 (découpage d'une chaîne en fragments) :

```
def decoupe(ch, n):
    "découpage de la chaîne ch en une liste de fragments de n caractères"
    d, f = 0, n             # indices de début et de fin de fragment
    tt = []                 # liste à construire
    while d < len(ch):
        if f > len(ch):     # on ne peut pas découper au-delà de la fin
            f = len(ch)
        fr = ch[d:f]        # découpage d'un fragment
        tt.append(fr)       # ajout du fragment à la liste
        d, f = f, f +n      # indices suivants
    return tt

def inverse(tt):
    "rassemble les éléments de la liste tt dans l'ordre inverse"
    ch = ""                 # chaîne à construire
    i = len(tt)             # on commence par la fin de la liste
    while i > 0 :
        i = i - 1           # le dernier élément possède l'indice n -1
        ch = ch + tt[i]
    return ch

# Test :
ch ="abcdefghijklmnopqrstuvwxyz123456789"
liste = decoupe(ch, 5)
print liste
print inverse(liste)
```

Exercice 106 :

```
# Rechercher l'indice d'un caractère dans une chaîne

def trouve(ch, car, deb=0):
    "trouve l'indice du caractère car dans la chaîne ch"
    i = deb
    while i < len(ch):
        if ch[i] == car:
            return i        # le caractère est trouvé -> on termine
        i = i + 1
    return -1               # toute la chaîne a été scannée sans succès

# Tests :
print trouve("Coucou c'est moi", "z")
print trouve("Juliette & Roméo", "&")
print trouve("César & Cléopâtre", "r", 5)
```

Exercice 109 :

```
prefixes, suffixe = "JKLMNOP", "ack"

for p in prefixes:
    print p + suffixe
```

Exercice 110 :

```
def compteMots(ch):
    "comptage du nombre de mots dans la chaîne ch"
    if len(ch) ==0:
        return 0
    nm = 1                  # la chaîne comporte au moins un mot
    for c in ch:
        if c == " ":        # il suffit de compter les espaces
            nm = nm + 1
    return nm

# Test :
print compteMots("Les petits ruisseaux font les grandes rivières")
```

Exercice 111 :

```
def majuscule(car):
    "retourne <vrai> si car est une majuscule"
    if car in "ABCDEFGHIJKLMNOPQRSTUVWXYZ":
        return 1
    else:
        return 0
```

Exercice 113 :

```
def chaineListe(ch):
    "convertit la chaîne ch en une liste de mots"
    liste, ct = [], ""          # ct est une chaîne temporaire
    for c in ch:
        if c == " ":
            liste.append(ct)    # ajouter la ch. temporaire à la liste
            ct = ""             # ré-initialiser la ch. temporaire
        else:
            ct = ct + c
    if ct != "":
        liste.append(ct)        # ne pas oublier le dernier mot
    return liste

# Test :
print chaineListe("Une hirondelle ne fait pas le printemps")
print chaineListe("")
```

Exercice 114 (utilise les deux fonctions définies dans les exercices précédents) :

```
txt = "Le nom de ce Monsieur est Alphonse"
lst = chaineListe(txt)          # convertir la phrase en une liste de mots
for mot in lst:                 # analyser chacun des mots de la liste
    if majuscule(mot[0]):       # tester le premier caractère du mot
        print mot
```

Exercice 115 :

```
def majuscule(car):
    "retourne <vrai> si car est une majuscule"
    if car >= "A" and car <= "Z":
        return 1
    else:
        return 0

def minuscule(car):
    "retourne <vrai> si car est une minuscule"
    if car >= "a" and car <= "z":
        return 1
    else:
        return 0

def alphab(car):
    "retourne <vrai> si car est un caractère alphabétique"
    if majuscule(car) or minuscule(car):
        return 1
    else:
        return 0
```

Exercice 118 (utilise deux fonctions définies dans les exercices précédents) :

```
def compteMaj(ch):
    "comptage des mots débutant par une majuscule dans la chaîne ch"
    c = 0
    lst = chaineListe(ch)       # convertir la phrase en une liste de mots
    for mot in lst:             # analyser chacun des mots de la liste
        if majuscule(mot[0]):   # tester le premier caractère du mot
            c = c +1
    return c

# Test :
print compteMaj("Les filles Tidgout se nomment Justine et Corinne")
```

Exercice 119 (table des codes ASCII) :

```
c = 32      # Premier code ASCII <imprimable>

while c < 128 :                 # caractères non accentués seulement
    print "Code", c, ":", chr(c), "  ",
    c = c + 1
```

Exercice 121 (Convertir majuscules -> minuscules et inversément) :

```
def convMajMin(ch):
    "échange les majuscules et les minuscules dans la chaîne ch"
    nouvC = ""                   # chaîne à construire
    for car in ch:
        code = ord(car)
        if car >= "A" and car <= "Z":
            code = code + 32
        elif car >= "a" and car <= "z":
            code = code - 32
        nouvC = nouvC + chr(code)
    return nouvC

# Test :
print convMajMin("Ferdinand-Charles de CAMARET")
```

Exercice 123 (Comptage de voyelles) :

```
def voyelle(car):
    "teste si car est une voyelle"
    if car in "AEIOUYaeiouy":
        return 1
    else:
        return 0

def compteVoyelles(ch):
    "compte les voyelles présentes dans la chaîne ch"
    n = 0
    for c in ch:
        if voyelle(c):
            n = n + 1
    return n

# Test :
print compteVoyelles("Monty Python Flying Circus")
```

Exercice 125 :

```
# Comptage du nombre de mots dans un texte

fiSource = raw_input("Nom du fichier à traiter : ")
fs = open(fiSource, 'r')

n = 0           # variable compteur
while 1:
    ch = fs.readline()
    if ch == "":
        break
    # conversion de la chaîne lue en une liste de mots :
    li = ch.split()
    # totalisation des mots :
    n = n + len(li)
fs.close()
print "Ce fichier texte contient un total de %s mots" % (n)
```

Exercice 126 :

```
# Conversion en majuscule du premier caractère de chaque ligne

fiSource = raw_input("Nom du fichier à traiter : ")
fiDest = raw_input("Nom du fichier destinataire : ")
fs = open(fiSource, 'r')
fd = open(fiDest, 'w')

while 1:
    ch = fs.readline()
    if ch == "":
        break
    if ch[0] >= "A" and ch[0] <= "Z":
        # le premier car. est une majuscule. On passe.
        pass
    else:
        # Reconstruction de la chaîne:
        pc = ch[0].upper()      # Premier caractère converti
        rc = ch[1:]             # toute le reste de la chaîne
        ch = pc + rc            # fusion
        # variante utilisant une méthode encore plus intégrée :
        # ch = ch.capitalize()
    # Transcription :
    fd.write(ch)

fd.close()
fs.close()
```

Exercice 127 :

```
# Fusion de lignes pour former des phrases

fiSource = raw_input("Nom du fichier à traiter : ")
fiDest = raw_input("Nom du fichier destinataire : ")
fs = open(fiSource, 'r')
fd = open(fiDest, 'w')
```

```
# On lit d'abord la première ligne :
ch1 = fs.readline()
# On lit ensuite les suivantes, en les fusionnant si nécessaire :
while 1:
    ch2 = fs.readline()
    if ch2 == "":
        break
    # Si la chaîne lue commence par une majuscule, on transcrit
    # la précédente dans le fichier destinataire, et on la
    # remplace par celle que l'on vient de lire :
    if ch2[0] >= "A" and ch2[0] <= "Z":
        fd.write(ch1)
        ch1 = ch2
    # Sinon, on la fusionne avec la précédente :
    else:
        ch1 = ch1[:-1] + " " + ch2
        # (veiller à enlever de ch1 le caractère de fin de ligne)

fd.write(ch1)        # ne pas oublier de transcrire la dernière !
fd.close()
fs.close()
```

Exercice 128 (caractéristiques de sphères) :

```
# Le fichier de départ est un fichier <texte> dont chaque ligne contient
# un nombre réel (encodé sous la forme d'une chaîne de caractères)

from math import pi

def caractSphere(d):
    "renvoie les caractéristiques d'une sphère de diamètre d"
    d = float(d)        # conversion de l'argument (=chaîne) en réel
    r = d/2             # rayon
    ss = pi*r**2        # surface de section
    se = 4*pi*r**2      # surface extérieure
    v = 4/3*pi*r**3     # volume
    # Le marqueur de conversion %8.2f utilisé ci-dessous formate le nombre
    # affiché de manière à occuper 8 caractères au total, en arrondissant
    # de manière à conserver deux chiffres après la virgule :
    ch = "Diam. %6.2f cm Section = %8.2f cm² " % (d, ss)
    ch = ch +"Surf. = %8.2f cm². Vol. = %9.2f cm³" % (se, v)
    return ch

fiSource = raw_input("Nom du fichier à traiter : ")
fiDest = raw_input("Nom du fichier destinataire : ")
fs = open(fiSource, 'r')
fd = open(fiDest, 'w')
while 1:
    diam = fs.readline()
    if diam == "" or diam == "\n":
        break
    fd.write(caractSphere(diam) + "\n")         # enregistrement
fd.close()
fs.close()
```

Exercice 129 :

```
# Mise en forme de données numériques
# Le fichier traité est un fichier <texte> dont chaque ligne contient un nombre
# réel (sans exposants et encodé sous la forme d'une chaîne de caractères)

def arrondir(reel):
    "représentation arrondie à .0 ou .5 d'un nombre réel"
    ent = int(reel)             # partie entière du nombre
    fra = reel - ent            # partie fractionnaire
    if fra < .25 :
        fra = 0
    elif fra < .75 :
        fra = .5
    else:
        fra = 1
    return ent + fra

fiSource = raw_input("Nom du fichier à traiter : ")
fiDest = raw_input("Nom du fichier destinataire : ")
fs = open(fiSource, 'r')
fd = open(fiDest, 'w')
while 1:
    ligne = fs.readline()
    if ligne == "" or ligne == "\n":
        break
    n = arrondir(float(ligne))      # conversion en <float>, puis arrondi
    fd.write(str(n) + "\n")         # enregistrement

fd.close()
fs.close()
```

Exercice 132 :

```
# Affichage de tables de multiplication

nt = [2, 3, 5, 7, 9, 11, 13, 17, 19]

def tableMulti(m, n):
     "renvoie n termes de la table de multiplication par m"
     ch =""
     for i in range(n):
          v = m * (i+1)              # calcul d'un des termes
          ch = ch + "%4d" % (v)      # formatage à 4 caractères
     return ch

for a in nt:
     print tableMulti(a, 15)         # 15 premiers termes seulement
```

Exercice 133 (simple parcours d'une liste) :

```
lst = ['Jean-Michel', 'Marc', 'Vanessa', 'Anne',
       'Maximilien', 'Alexandre-Benoît', 'Louise']

for e in lst:
     print "%s : %s caractères" % (e, len(e))
```

Exercice 134 :

```
# Elimination de doublons

lst = [9, 12, 40, 5, 12, 3, 27, 5, 9, 3, 8, 22, 40, 3, 2, 4, 6, 25]
lst2 = []

for el in lst:
     if el not in lst2:
          lst2.append(el)

lst2.sort()
print lst2
```

Exercice 136 (afficher tous les jours d'une année) :

```
## Cette variante utilise une liste de listes ##
## (que l'on pourrait aisément remplacer par deux listes distinctes)

# La liste ci-dessous contient deux éléments qui sont eux-mêmes des listes.
# l'élément 0 contient les nombres de jours de chaque mois, tandis que
# l'élément 1 contient les noms des douze mois :
mois = [[31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31],
        ['Janvier', 'Février', 'Mars', 'Avril', 'Mai', 'Juin', 'Juillet',
         'Août', 'Septembre', 'Octobre', 'Novembre', 'Décembre']]

jour = ['Dimanche','Lundi','Mardi','Mercredi','Jeudi','Vendredi','Samedi']

ja, jm, js, m = 0, 0, 0, 0

while ja <365:
    ja, jm = ja +1, jm +1    # ja = jour dans l'année, jm = jour dans le mois
    js = (ja +3) % 7         # js = jour de la semaine. Le décalage ajouté
                             #      permet de choisir le jour de départ

    if jm > mois[0][m]:               # élément m de l'élément 0 de la liste
        jm, m = 1, m+1

    print jour[js], jm, mois[1][m]    # élément m de l'élément 1 de la liste
```

Exercice 139 :

```
# Insertion de nouveaux éléments dans une liste existante

t1 = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
t2 = ['Janvier','Février','Mars','Avril','Mai','Juin',
      'Juillet','Août','Septembre','Octobre','Novembre','Décembre']

c, d = 1, 0
while d < 12 :
     t2[c:c] = [t1[d]]       # ! l'élément inséré doit être une liste
     c, d = c+2, d+1
```

Exercice 143 :

```
# Crible d'Eratosthène pour rechercher les nombres premiers de 1 à 999

# Créer une liste de 1000 éléments 1 (leurs indices vont de 0 à 999) :
lst = [1]*1000
# Parcourir la liste à partir de l'élément d'indice 2:
for i in range(2,1000):
    # Mettre à zéro les éléments suivants dans la liste,
    # dont les indices sont des multiples de i :
    for j in range(i*2, 1000, i):
        lst[j] = 0

# Afficher les indices des éléments restés à 1 (on ignore l'élément 0) :
for i in range(1,1000):
    if lst[i]:
        print i,
```

Exercice 146 (Test du générateur de nombres aléatoires, page 137) :

```
from random import random           # tire au hasard un réel entre 0 et 1
from string import atoi             # convertit une chaîne en nombre entier

n = raw_input("Nombre de valeurs à tirer au hasard (défaut = 1000) : ")
if n == "":
    nVal =1000
else:
    nVal = atoi(n)

n = raw_input("Nombre de fractions dans l'intervalle 0-1 (entre 2 et "
              + str(nVal/10) + ", défaut =5) : ")
if n == "":
    nFra =5
else:
    nFra = atoi(n)

if nFra < 2:
    nFra =2
elif nFra > nVal/10:
    nFra = nVal/10

print "Tirage au sort des", nVal, "valeurs ..."
listVal = [0]*nVal                      # créer une liste de zéros
for i in range(nVal):                   # puis modifier chaque élément
    listVal[i] = random()

print "Comptage des valeurs dans chacune des", nFra, "fractions ..."
listCompt = [0]*nFra                    # créer une liste de compteurs

# parcourir la liste des valeurs :
for valeur in listVal:
    # trouver l'index de la fraction qui contient la valeur :
    index = int(valeur*nFra)
    # incrémenter le compteur correspondant :
    listCompt[index] = listCompt[index] +1

# afficher l'état des compteurs :
for compt in listCompt:
    print compt,
```

Exercice 147 : tirage de cartes

```
from random import randrange

couleurs = ['Pique', 'Trèfle', 'Carreau', 'Coeur']
valeurs = [2, 3, 4, 5, 6, 7, 8, 9, 10, 'valet', 'dame', 'roi', 'as']

# Construction de la liste des 52 cartes :
carte =[]
for coul in couleurs:
     for val in valeurs:
          carte.append("%s de %s" % (str(val), coul))

# Tirage au hasard :
while 1:
     k = raw_input("Frappez <c> pour tirer une carte, <Enter> pour terminer ")
     if k =="":
          break
     r = randrange(52)
     print carte[r]
```

Exercice 148 : utilisation d'un dictionnaire

```
dico = {'Marinette':(15, 1.65),
        'Sylvie':(14, 1.58),
        'Albert':(16, 1.72),
        'Juliette':(15, 1.59),
        'Nicolas':(14, 1.62)}

for cle in dico:
    item = dico[cle]
    age = item[0]
    taille  = item[1]
    print "Nom = %s - âge = %s ans - taille = %s m." % (cle, age, taille)
```

Exercice 149 : échange des clés et des valeurs dans un dictionnaire

```
def inverse(dico):
    "Construction d'un nouveau dico, pas à pas"
    dic_inv ={}
    for cle in dico:
        item = dico[cle]
        dic_inv[item] = cle

    return dic_inv

# programme test :

dico = {'Computer':'Ordinateur',
        'Mouse':'Souris',
        'Keyboard':'Clavier',
        'Hard disk':'Disque dur',
        'Screen':'Ecran'}

print dico
print inverse(dico)
```

Exercice 150 : histogramme

```
nFich = raw_input('Nom du fichier : ')
fi = open(nFich, 'r')
texte = fi.read()       # conversion du fichier en une chaîne de caractères
fi.close()

print texte
dico ={}
for c in texte:
    c = c.upper()       # conversion de toutes les lettres en majuscules
    dico[c] = dico.get(c, 0) +1

liste = dico.items()
liste.sort()
print liste
```

Exercice 151 :

```
nFich = raw_input('Nom du fichier à traiter : ')
fi = open(nFich, 'r')
texte = fi.read()
fi.close()

# afin de pouvoir aisément séparer les mots du texte, on commence
# par convertir tous les caractères non-alphabétiques en espaces  :

alpha = "abcdefghijklmnopqrstuvwxyzéèàùçâêîôûäëïöü"

lettres = ''            # nouvelle chaîne à construire
for c in texte:
    c = c.lower()       # conversion de chaque caractère en minuscule
    if c in alpha:
        lettres = lettres + c
    else:
        lettres = lettres + ' '

# conversion de la chaîne résultante en une liste de mots :
mots = lettres.split()

# construction de l'histogramme :
dico ={}
for m in mots:
    dico[m] = dico.get(m, 0) +1

liste = dico.items()

# tri de la liste résultante :
liste.sort()

# affichage en clair :
for item in liste:
    print item[0], ":", item[1]
```

Exercice 153 :

```
class Domino:
    def __init__(self, pa, pb):
        self.pa, self.pb = pa, pb

    def affiche_points(self):
        print "face A :", self.pa,
        print "face B :", self.pb

    def valeur(self):
        return self.pa + self.pb

# Programme de test :

d1 = Domino(2,6)
d2 = Domino(4,3)

d1.affiche_points()
d2.affiche_points()

print "total des points :", d1.valeur() + d2.valeur()

liste_dominos = []
for i in range(7):
    liste_dominos.append(Domino(6, i))

vt =0
for i in range(7):
    liste_dominos[i].affiche_points()
    vt = vt + liste_dominos[i].valeur()

print "valeur totale des points", vt
```

Exercice 155 :

```
class Voiture:
    def __init__(self, marque = 'Ford', couleur = 'rouge'):
        self.couleur = couleur
        self.marque = marque
        self.pilote = 'personne'
        self.vitesse = 0

    def accelerer(self, taux, duree):
        if self.pilote =='personne':
            print "Cette voiture n'a pas de conducteur !"
        else:
            self.vitesse = self.vitesse + taux * duree

    def choix_conducteur(self, nom):
        self.pilote = nom

    def affiche_tout(self):
            print "%s %s pilotée par %s, vitesse = %s m/s" % \
            (self.marque, self.couleur, self.pilote, self.vitesse)

a1 = Voiture('Peugeot', 'bleue')
a2 = Voiture(couleur = 'verte')
a3 = Voiture('Mercedes')
a1.choix_conducteur('Roméo')
a2.choix_conducteur('Juliette')
a2.accelerer(1.8, 12)
a3.accelerer(1.9, 11)
a2.affiche_tout()
a3.affiche_tout()
```

Exercice 156 :

```
class Satellite:
    def __init__(self, nom, masse =100, vitesse =0):
        self.nom, self.masse, self.vitesse = nom, masse, vitesse

    def impulsion(self, force, duree):
        self.vitesse = self.vitesse + force * duree / self.masse

    def energie(self):
        return self.masse * self.vitesse**2 / 2

    def affiche_vitesse(self):
        print "Vitesse du satellite %s = %s m/s" \
                           % (self.nom, self.vitesse)

# Programme de test :

s1 = Satellite('Zoé', masse =250, vitesse =10)

s1.impulsion(500, 15)
s1.affiche_vitesse()
print s1.energie()
s1.impulsion(500, 15)
s1.affiche_vitesse()
print s1.energie()
```

Exercices 157-158 (classes de cylindres et de cônes) :

```
# Classes dérivées - polymorphisme

class Cercle:
    def __init__(self, rayon):
        self.rayon = rayon

    def surface(self):
        return 3.1416 * self.rayon**2

class Cylindre(Cercle):
    def __init__(self, rayon, hauteur):
        Cercle.__init__(self, rayon)
        self.hauteur = hauteur

    def volume(self):
        return self.surface()*self.hauteur

        # la méthode surface() est héritée de la classe parente

class Cone(Cylindre):
    def __init__(self, rayon, hauteur):
        Cylindre.__init__(self, rayon, hauteur)

    def volume(self):
        return Cylindre.volume(self)/3

        # cette nouvelle méthode volume() remplace celle que
        # l'on a héritée de la classe parente (exemple de polymorphisme)

cyl = Cylindre(5, 7)
print cyl.surface()
print cyl.volume()

co = Cone(5,7)
print co.surface()
print co.volume()
```

Exercice 159 :

```
# Tirage de cartes

from random import randrange

class JeuDeCartes:
    """Jeu de cartes"""
    # attributs de classe (communs à toutes les instances) :
    couleur = ('Pique', 'Trèfle', 'Carreau', 'Coeur')
    valeur = (0, 0, 2, 3, 4, 5, 6, 7, 8, 9, 10, 'valet', 'dame', 'roi', 'as')

    def __init__(self):
        "Construction de la liste des 52 cartes"
        self.carte =[]
        for coul in range(4):
            for val in range(13):
                self.carte.append((val +2, coul))   # la valeur commence à 2

    def nom_carte(self, c):
        "Renvoi du nom de la carte c, en clair"
        return "%s de %s" % (self.valeur[c[0]], self.couleur[c[1]])

    def battre(self):
        "Mélange des cartes"
        t = len(self.carte)             # nombre de cartes restantes
        # pour mélanger, on procède à un nombre d'échanges équivalent :
        for i in range(t):
            # tirage au hasard de 2 emplacements dans la liste :
            h1, h2 = randrange(t), randrange(t)
            # échange des cartes situées à ces emplacements :
            self.carte[h1], self.carte[h2] = self.carte[h2], self.carte[h1]

    def tirer(self):
        "Tirage de la première carte de la pile"
        t = len(self.carte)             # vérifier qu'il reste des cartes
        if t >0:
            carte = self.carte[0]       # choisir la première carte du jeu
            del(self.carte[0])          # la retirer du jeu
            return carte                # en renvoyer copie au prog. appelant
        else:
            return None                 # facultatif


### Programme test :

if __name__ == '__main__':
    jeu = JeuDeCartes()                 # instanciation d'un objet
    jeu.battre()                        # mélange des cartes
    for n in range(53):                 # tirage des 52 cartes :
        c = jeu.tirer()
        if c == None:                   # il ne reste aucune carte
            print 'Terminé !'           # dans la liste
        else:
            print jeu.nom_carte(c)      # valeur et couleur de la carte
```

Exercice 160 :

(On supposera que l'exercice précédent a été sauvegardé sous le nom cartes.py)

```
# Bataille de de cartes

from cartes import JeuDeCartes

jeuA = JeuDeCartes()        # instanciation du premier jeu
jeuB = JeuDeCartes()        # instanciation du second jeu
jeuA.battre()               # mélange de chacun
jeuB.battre()
pA, pB = 0, 0               # compteurs de points des joueurs A et B

# tirer 52 fois une carte de chaque jeu :
for n in range(52):
    cA, cB = jeuA.tirer(), jeuB.tirer()
    vA, vB = cA[0], cB[0]   # valeurs de ces cartes
    if vA > vB:
        pA += 1
    elif vB > vA:
        pB += 1             # (rien ne se passe si vA = vB)
    # affichage des points successifs et des cartes tirées :
     print "%s * %s ==> %s * %s" % (jeuA.nom_carte(cA), jeuB.nom_carte(cB), pA,
pB)

print "le joueur A obtient %s points, le joueur B en obtient %s." % (pA, pB)
```

Exercice 180 (Création de la base de données "musique") :

```
import gadfly

connex = gadfly.gadfly()
connex.startup("musique","C:\Python20\essais\gadfly")
cur = connex.cursor()
requete = "create table compositeurs (comp varchar, a_naiss integer,\
           a_mort integer)"
cur.execute(requete)
requete = "create table oeuvres (comp varchar, titre varchar,\
           duree integer, interpr varchar)"
cur.execute(requete)

print "Entrée des enregistrements, table des compositeurs :"
while 1:
    nm = raw_input("Nom du compositeur (<Enter> pour terminer) : ")
    if nm =='':
        break
    an = raw_input("Année de naissance : ")
    am = raw_input("Année de mort : ")
    requete ="insert into compositeurs(comp, a_naiss, a_mort) values \
                 ('%s', %s, %s)" % (nm, an, am)
    cur.execute(requete)
# Affichage des données entrées, pour vérification :
cur.execute("select * from compositeurs")
print cur.pp()

print "Entrée des enregistrements, table des oeuvres musicales :"
while 1:
    nom = raw_input("Nom du compositeur (<Enter> pour terminer) : ")
    if nom =='':
        break
    tit = raw_input("Titre de l'oeuvre : ")
    dur = raw_input("durée (minutes) : ")
    int = raw_input("interprète principal : ")
    requete ="insert into oeuvres(comp, titre, duree, interpr) values \
                 ('%s', '%s', %s, '%s')" % (nom, tit, dur, int)
    cur.execute(requete)
# Affichage des données entrées, pour vérification :
cur.execute("select * from oeuvres")
print cur.pp()

connex.commit()
```

0.2 Quelques idées de projets à développer :

Jeu de "bataille" aux cartes - Jeu de "Master Mind"

Jeu d'Othello (Reversi) – l'ordinateur ne joue pas, mais se contente de rafraîchir l'affichage après chaque coup joué par l'un des deux opposants (+ respect des règles).

Simulation du jeu de la vie de Conway

Jeu de dominos

Jeu de "Puissance 4" - Jeu de 21 - Jeu "Des chiffres et des lettres"

Jeu de solitaire - Jeu du serpent

Jeu de bataille navale

Panneau électrique

Gestion de cave à vins

Simulation de coupe du monde de football

Gestion des équipes dans un tournoi de cartes, une lan-party, etc.

Convertisseur de code Morse

Traceur de fonctions mathématiques

Algorithmes de tri

Poésie automatique

Cryptage et décryptage

Simulation de phénomènes physiques divers (gravitation, calorimètre, etc)

Horloge analogique

Traitement d'images

Calculatrice graphique

Calcul du coût de communications téléphoniques diverses

Scoring : intérêts bancaires, octroi d'un prêt

Calcul des primes d'assurance RC voiture

Calcul d'intégrales

0.3 Annexes tirées du texte original "How to think like a computer scientist"

Suivant les termes de la *GNU Free Documentation licence (voir p. 278)*, les annexes qui suivent doivent obligatoirement accompagner telles quelles toute distribution du texte original, que celui-ci ait été modifié (traduit, par exemple) ou non.

0.3.1 Preface

by J. Elkner

This book owes its existance to the collaboration made possible by the internet and the free software movement. Its three authors, a college professor, a high school teacher, and a professional programmer, have yet to meet face to face, but we have been able to work closely together, and have been aided by many wonderful folks who have donated their time and energy to helping make it better.

What excites me most about it is that it is a testament to both the benefits and future possibilities of this kind of collaboration, the framework for which has been put in place by Richard Stallman and the Free Software Foundation.

a) How and why I came to use Python

In 1999, the College Board's Advanced Placement Computer Science exam was given in C++ for the first time. As in many high schools throughout the country, the decision to change languages had a direct impact on the computer science curriculum where I teach at Yorktown High School, in Arlington, Virginia. Up to this point, Pascal was the language of instruction in both our first year and AP courses. In keeping with past practice of giving students two years of exposure to the same language, we made the decision to switch to C++ in the first year course for the 1997-98 school year, so that we would be in step with the College Board's change for the AP course the following year.

Two years later, I was convinced that C++ was a poor choice to use for introducing students to computer science. While it is certainly a very powerful programming language, it is also an extremely difficult language to learn and teach. I found myself constantly fighting with C++'s difficult syntax and multiple ways of doing things, and I was losing many students unnecessarily as a result. Convinced there had to be a better language choice for our first year class, I went looking for an alternative to C++.

A discussion on the High School Linux Users' Group mailing list provided a solution. A thread emerged during the latter part of January, 1999 concerning the best programming language for use with first time high school computer science students. In a posting on January 30th, Brendon Ranking wrote:

I believe that Python is the best choice for any entry-level programming class. It teaches proper programming principles while being incredibly easy to learn. It is also designed to be object oriented from its inception so it doesn't have the add-on pain that both Perl and C++ suffer from...... It is also *very* widely supported and very much web-centric, as well.

I had first heard of Python a few years earlier at a Linux Install Fest, when an enthusiastic Michael McLay told me about Python's many merits. He and Brendon had now convinced me that I needed to look into Python.

Matt Ahrens, one of Yorktown's gifted students, jumped at the chance to try out Python, and in the final two months of the 1998-99 school year he not only learned the language but wrote an application called pyTicket which enabled our staff to report technology problems via the web. I knew that Matt could not have finished an application of that scale in so short a time in C++, and this accomplishment combined with Matt's positive assessment of Python suggested Python was the solution I was looking for.

b) Finding a text book

Having decided to use Python in both my introductory computer science classes the following year, the most pressing problem was the lack of an available text book.

Free content came to the rescue. Earlier in the year Richard Stallman had introduced me to Allen Downey. Both of us had written to Richard expressing an interest in developing free educational content. Allen had already written a first year computer science text book titled, **How to think like a computer**

scientist. When I read this book I knew immediately that I wanted to use it in my class. It was the clearest and most helpful computer science text I had seen. It emphasized the processes of thought involved in programming, rather than the features of a particular language. Reading it immediately made me a better teacher.

Not only was How to think like a computer scientist an excellent book, but it was also released under a GNU public license, which meant it could be used freely and modified to meet the needs of its user. Once I decided to use Python, it occurred to me that I could translate Allen's original Java version into the new language. While I would not have been able to write a text book on my own, having Allen's book to work from made it possible for me to do so, at the same time demonstrating that the cooperative development model used so well in software could also work for educational content.

Working on this book for the last two years has been rewarding for both me and my students, and the students played a big part in the process. Since I could make instant changes whenever someone found a spelling error or difficult passage, I encouraged them to look for errors in the book by giving them a bonus point every time they found or suggested something that resulted in a change in the text. This had the double benefit of encouraging them to read the text more carefully, and of getting the text thoroughly reviewed by its most important critics, students using it to learn computer science.

For the second half of the book on object oriented programming, I knew that someone with more real programming experience than I had would be needed to do it right. The book actually sat in an unfinished state for the better part of a year until two things happened that led to its completion.

I received an email from Chris Meyers expressing interest in the book. Chris is a professional programmer who started teaching a programming course last year using Python at Lane Community College in Eugene Oregon. The prospect of teaching the course had led Chris to the book, and he started helping out with it immediately. By the end of the school year he had created a companion project on our web site at http://www.ibiblio.org/obp called Python for Fun and was working with some of my most advanced students as a master teacher, guiding them beyond the places I could take them.

c) Introducing programming with Python

The process of translating and using How to think like a computer scientist for the past two years has confirmed Python's suitability to teaching beginning students. Python greatly simplifies programming examples and makes important programming ideas easier to teach.

The first example from the text dramatically illustrates this point. It is the traditional "hello, world" program, which in the C++ version of the book looks like this:

```
#include <iostream.h>
void main()
{
     cout << "Hello, world." << endl;
}
```

in the Python version it becomes:

```
print "Hello, World!"
```

Even though this is a trivial example, the advantages to Python stand out. There are no prerequisites to Yorktown's Computer Science I course, so many of the students seeing this example are looking at their first program. Some of them are undoubtedly a little nervous, having heard that computer programming is difficult to learn. The C++ version has always forced me to choose between two unsatisfying options: either to explain the #include, void main(), {, and } statements, and risk confusing or intimidating some of the students right at the start, or to tell them "just don't worry about all of that stuff now, we will talk about it later" and risk the same thing. The educational objectives at this point in the course are to introduce students to the idea of a programming statement and to get them to make their first program, thereby introducing them to the programming environment. The Python program has exactly what is needed to do these things, and nothing more.

Comparing Section 1.5 of each version of the book, where this first program is located, further illustrates what this means to the beginning student. There are thirteen paragraphs of explanation of "Hello, world" in the C++ version, in the Python version there are only two. More importantly, the missing eleven paragraphs do not deal with the "big ideas" in computer programming, but with the minutia of C++ syntax. I found this same thing happening throughout the book. Whole paragraphs simply disappear from the Python version of the text because Python's much clearer syntax renders them unnecessary.

Using a very high level language like Python allows a teacher to postpone talking about low level details of the machine until students have the background that they need to better make sense of the details. It thus creates the ability to put "first things first" pedagogically.

One of the best examples of this is the way in which Python handles variables. In C++ a variable is a name for a place which holds a thing. Variables have to be declared with types at least in part because the size of the place to which they refer needs to be predetermined. Thus the idea of a variable is bound up with the hardware of the machine. The powerful and fundamental concept of a variable is already difficult enough for beginning students (in both Computer Science and Algebra). Bytes and addresses do not help the matter. In Python a variable is a name which refers to a thing. This is a far more intuitive concept for beginning students, and one which is much closer to the meaning of variable that they learned in their math class. I had much less difficulty teaching variables this year than I did in the past, and I spent less time helping students with problems using them.

Another example of how Python aides in the teaching and learning of programming is in its syntax for functions. My students have always had a great deal of difficulty understanding functions. The main problem centers around the difference between a function definition and a function call, and the related distinction between a parameter and an argument. Python comes to the rescue with syntax that is nothing short of beautiful. Function definitions begin with the key word def, so I simply tell my students, "when you define a function, begin with def, followed by the name of the function that you are defining, when you call a function, simply call (type) out its name." Parameters go with definitions, arguments go with calls. There are no return types or parameter types or reference and value parameters to get in the way, so I am now able to teach functions in less then half the time that it previously took me, with better comprehension.

Using Python has improved the effectiveness of our computer science program for all students. I see a higher general level of success and a lower level of frustration than I experienced during the two years using C++. I move faster with better results. More students leave the course with the ability to create meaningful programs, and with the positive attitude toward the experience of programming that this engenders.

d) Building a community

I have received email every continent on the globe and from as far away as Korea from people using this book to learn or to teach programming. A user community has begun to emerge and increasing numbers of people have been contributing to the project by sending in materials for the companion web site at http://www.ibiblio.org/obp.

With the publication of the book in print form, I expect the growth in the user community to continue and accelerate. It is the emergence of this user community and the possibility it suggests for similar collaboration among educators that has been the most exciting thing for me about working on the project. By working together we can both increase the quality of materials available for our use and save valuable time. I invite you to join our community and look forward to hearing from you.

Jeffrey Elkner
Yorktown High School
Arlington, Virginia

0.3.2 Contributor list

by Jeffrey Elkner

Perhaps the most exciting thing about a free content textbook is the possibility it creates for those using the book to collaborate in its development. I have been delighted by the many responses, suggestions, corrections, and words of encouragement I have received from people who have found this book to be useful, and who have taken the time to let me know about it.

Unfortunately, as a busy high school teacher who is working on this project in my spare time (what little there is of it ;-), I have been neglectful in giving credit to those who have helped with the book. I always planned to add an "Acknowlegdements" sections upon completion of the first stable version of the book, but as time went on it became increasingly difficult to even track those who had contributed.

Upon seeing the most recent version of Tony Kuphaldt's wonderful free text, "Lessons in Electric Circuits", I got the idea from him to create an ongoing "Contributor List" page which could be easily modified to include contributors as they come in.

My only regret is that many earlier contributors might be left out. I will begin as soon as possible to go back through old emails to search out the many wonderful folks who have helped me in this endeavour. In the mean time, if you find yourself missing from this list, please accept my humble apologies and drop me an email at jeff@elkner.net to let me know about my oversight.

And so, without further delay, here is a listing of the contributors:

Lloyd Hugh Allen

Lloyd sent in a correction to section 8.4. He can be reached at: lha2@columbia.edu

Yvon Boulianne

Yvon sent in a correction of a logical error in Chapter 5. She can be reached at: mystic@monuniverse.net

Fred Bremmer

Fred submitted a correction in section 2.1. He can be reached at: Fred.Bremmer@ubc.cu

Jonah Cohen

Jonah wrote the Perl scripts to convert the LaTeX source for this book into beautiful HTML. His Web page is jonah.ticalc.org and his email is JonahCohen@aol.com

Michael Conlon

Michael sent in a grammer correction in Chapter 2 and an improvement in style in Chapter 1, and he initiated discussion on the technical aspects of interpreters. His email is michael.conlon@sru.edu

Courtney Gleason

Courtney and Katherine Smith created the first version of horsebet.py, which is used as the case study for the last chapters of the book. Courtney can be reached at: orion1558@aol.com

Lee Harr

Lee submitted corrections for sections 10.1 and 11.5. He can be reached at: missive@linuxfreemail.com

James Kaylin

James is a student using the text. He has submitted numerous corrections. His email is Jamarf@aol.com

David Kershaw

David fixed the broken catTwice function in section 3.10. His email is david_kershaw@merck.com

Eddie Lam

Eddie has sent in numerous corrections to Chapters 1, 2, and 3. He also fixed the Makefile so that it creates an index the first time it is run and helped us set up a versioning scheme. Eddie can be reached at: nautilus@yoyo.cc.monash.edu.au

Man-Yong Lee

Man-Yong sent in a correction to the example code in section 2.4. His email is yong@linuxkorea.co.kr

David Mayo

While he didn't mean to hit us over the head with it, David Mayo pointed out that the word "unconsciously" in chapter 1 needed to be changed to "subconsciously". Email : bdbear44@netscape.net

Chris McAloon

Chris sent in several corrections to sections 3.9 and 3.10. He can be reached at: cmcaloon@ou.edu

Matthew J. Moelter

Matthew has been a long-time contributor who sent in numerous corrections and suggestions to the book. He can be reached at: mmoelter@calpoly.edu

Simon Dicon Montford

Simon reported a missing function definition and several typos in Chapter 3. He also found errors in the increment function in Chapter 13. He can be reached at: dicon@bigfoot.com

John Ouzts

John sent in a correction to the "return value" definition in Chapter 3. His email is jouzts@bigfoot.com

Kevin Parks

Kevin sent in valuable comments and suggestions as to how to improve the distribution of the book. He can be reached at: cpsoct@lycos.com

David Pool

David sent in a typo in the glossary of chapter 1, as well as kind words of encouragement. He can be reached at: pooldavid@hotmail.com

Michael Schmitt

Michael sent in a correction to the chapter on files and exceptions. He can be reached at: ipv6_128@yahoo.com

Paul Sleigh

Paul found an error in Chapter 7 and a bug in Jonah Cohen's Perl script that generates HTML from LaTeX. He can be reached at: bat@atdot.dotat.org

Christopher Smith

Chris is a computer science teacher at the Blake School in Minnesota who teaches Python to his beginning students. He can be reached at: csmith@blakeschool.org or smiles@saysomething.com

Katherine Smith

Katherine and Courtney Gleason created the first version of horsebet.py, which is used as the case study for the last chapters of the book. Katherine can be reached at: kss_0326@yahoo.com

Craig T. Snydal

Craig is testing the text in a course at Drew University. He has contributed several valuable suggestions and corrections, and can be reached at: csnydal@drew.edu

Ian Thomas

Ian and his students are using the text in a programming course. They are the first ones to test the chapters in the latter half of the book, and they have make numerous corrections and suggestions. Ian can be reached at: ithomas@sd70.bc.ca

Keith Verheyden

Keith made correction in Section 3.11 and can be reached at: kverheyd@glam.ac.uk

Chris Wrobel

Chris made corrections to the code in the chapter on file I/O and exceptions. ferz980@yahoo.com

Moshe Zadka

Moshe has made invaluable contributions to this project. In addition to writing the first draft of the chapter on Dictionaries, he provided continual guidance in the early stages of the book. He can be reached at: moshez@math.huji.ac.il

0.3.3 GNU Free Documentation License

Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft," which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

0.3.3.a.1 Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document," below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you."

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical, or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque."

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, \LaTeX~input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

0.3.3.a.2 Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in Section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

0.3.3.a.3 Copying in Quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

0.3.3.a.4 Modifications

You may copy and distribute a Modified Version of the Document under the conditions of Sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- State on the Title page the name of the publisher of the Modified Version, as the publisher.
- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- Include, immediately after the copyright notices, a license notice giving the public permission to use the

Modified Version under the terms of this License, in the form shown in the Addendum below.

- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- Include an unaltered copy of this License.
- Preserve the section entitled "History," and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- In any section entitled "Acknowledgements" or "Dedications," preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- Delete any section entitled "Endorsements." Such a section may not be included in the Modified Version.
- Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements," provided it contains nothing but endorsements of your Modified Version by various parties---for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

0.3.3.a.5 Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in Section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements," and any sections entitled "Dedications." You must delete all sections entitled "Endorsements."

0.3.3.a.6 Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

0.3.3.a.7 Aggregation with Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate," and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of Section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

0.3.3.a.8 Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of Section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

0.3.3.a.9 Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense, or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

0.3.3.a.10 Future Revisions of This License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See

http:///www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

0.3.3.a.11 Addendum: How to Use This License for Your Documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the licence is included in the section entitled "GNU Free Documentation License."

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Table des matières

Introduction 4
Choix d'un premier langage de programmation 5
Présentation du langage Python, par Stéfane Fermigier[2]. 6
Distribution de Python - Bibliographie 8
Pour le professeur : organisation du cours 9
Remerciements 10
Chapitre 1 : Penser comme un programmeur 11
1.1 L'activité de programmation 11
1.2 Langage machine, langage de programmation 11
1.3 Compilation et interprétation 13
1.4 Mise au point d'un programme - Recherche des erreurs ("debug") 15
1.4.1 Erreurs de syntaxe 15
1.4.2 Erreurs sémantiques 15
1.4.3 Erreurs à l'exécution 16
1.5 Recherche des erreurs et expérimentation 16
1.6 Langages naturels et langages formels 17
Chapitre 2 : Premières instructions 19
2.1 Calculer avec Python 19
2.2 Données et variables 21
2.3 Noms de variables et mots réservés 22
2.4 Affectation (ou assignation) 23
2.5 Afficher la valeur d'une variable 24
2.6 Typage des variables 24
2.7 Affectations multiples 25
2.8 Opérateurs et expressions 26
2.9 Priorité des opérations 27
2.10 Composition 28
Chapitre 3 : Contrôle du flux d'instructions 29
3.1 Suite (ou séquence[9]) d'instructions 29
3.2 Exécution conditionnelle 29
3.3 Opérateurs de comparaison 31
3.4 Instructions composées – Blocs d'instructions 31
3.5 Instructions imbriquées 32
3.6 Quelques règles de syntaxe Python 32
3.6.1 Les limites des instructions et des blocs sont définies par la mise en page 32
3.6.2 Instruction composée = En-tête , double point , bloc indenté 33
3.6.3 Les espaces et les commentaires sont normalement ignorés 33
Chapitre 4 : Instructions répétitives. 34
4.1 Ré-affectation 34
4.2 Répétitions en boucle - l'instruction while 35

4.3 Élaboration de tables 37
4.4 Construction d'une suite mathématique 37
4.5 Premiers scripts, ou : Comment conserver nos programmes ? 38
Chapitre 5 : Principaux types de données 41
5.1 Les données numériques 41
5.1.1 Les types "integer" et "long" 41
5.1.2 Le type "float" 43
5.2 Les données alphanumériques 45
5.2.1 Le type "string" (chaîne de caractères) 45
5.2.2 Accès aux caractères individuels d'une chaîne 46
5.2.3 Opérations élémentaires sur les chaînes 47
5.3 Les listes (première approche) 49
Chapitre 6 : Fonctions prédéfinies 52
6.1 Interaction avec l'utilisateur : la fonction input() 52
6.2 Importer un module de fonctions 53
6.3 Un peu de détente avec le module turtle 55
6.4 Véracité/fausseté d'une expression 56
6.5 Révision 57
6.5.1 Contrôle du flux - Utilisation d'une liste simple 57
6.5.2 Boucle while - Instructions imbriquées 58
6.5.3 Exercices 59
Chapitre 7 : Fonctions originales 61
7.1 Définir une fonction 61
7.1.1 Fonction simple sans paramètres 62
7.1.2 Fonction avec paramètre 64
7.1.3 Utilisation d'une variable comme argument 65
7.1.4 Fonction avec plusieurs paramètres 66
7.2 Variables locales, variables globales 67
7.3 "Vraies" fonctions et procédures 69
7.4 Utilisation des fonctions dans un script 71
7.5 Modules de fonctions 72
7.6 Typage des paramètres 76
7.7 Valeurs par défaut pour les paramètres 76
7.8 Arguments avec étiquettes 77
Chapitre 8 : Utilisation de fenêtres et de graphismes 79
8.1 Interfaces graphiques (GUI) 79
8.2 Premiers pas avec Tkinter 79
8.3 Programmes pilotés par des événements 83
8.3.1 Exemple graphique : tracé de lignes dans un canevas 85
8.3.2 Exemple graphique : calculatrice minimaliste 88
8.3.3 Exemple graphique : détection et positionnement d'un clic de souris 90
8.4 Les classes de widgets Tkinter 91

8.5 Utilisation de la méthode grid() pour contrôler la disposition des widgets 92
8.6 Composition d'instructions pour écrire un code plus compact 96
8.7 Modification des propriétés d'un objet - Animation 98
8.8 Animation automatique - Récursivité 101
Chapitre 9 : Les fichiers 104
9.1 Utilité des fichiers 104
9.2 Travailler avec des fichiers 105
9.3 Noms de fichiers - Répertoire courant 106
9.4 Les deux formes d'importation 106
9.5 Écriture séquentielle dans un fichier 108
9.6 Lecture séquentielle d'un fichier 109
9.7 L'instruction break pour sortir d'une boucle 110
9.8 Fichiers texte 111
9.9 Enregistrement et restitution de variables diverses 113
9.10 Gestion des exceptions. Les instructions try – except - else 114
Chapitre 10 : Approfondir les structures de données 117
10.1 Le point sur les chaînes de caractères 117
10.1.1 Concaténation, Répétition 117
10.1.2 Indiçage, extraction, longueur 117
10.1.3 Parcours d'une séquence. L'instruction for ... in ... .119
10.1.4 Appartenance d'un élément à une séquence. L'instruction in utilisée seule 120
10.1.5 Les chaînes sont des séquences non modifiables 121
10.1.6 Les chaînes sont comparables 122
10.1.7 Classement des caractères 122
10.1.8 Les chaînes sont des objets 124
10.1.9 Formatage des chaînes de caractères 126
10.2 Le point sur les listes 128
10.2.1 Définition d'une liste – Accès à ses éléments 128
10.2.2 Les listes sont modifiables 129
10.2.3 Les listes sont des objets 129
10.2.4 Techniques de "slicing" avancé pour modifier une liste 131
10.2.5 Création d'une liste de nombres à l'aide de la fonction range() 132
10.2.6 Parcours d'une liste à l'aide de for, range() et len() 132
10.2.7 Une conséquence du typage dynamique 133
10.2.8 Opérations sur les listes 133
10.2.9 Test d'appartenance 133
10.2.10 Copie d'une liste 134
10.2.11 Nombres aléatoires - Histogrammes 136
10.3 Les tuples 139
10.4 Les dictionnaires 140
10.4.1 Création d'un dictionnaire 140
10.4.2 Opérations sur les dictionnaires 141

10.4.3 Les dictionnaires sont des objets 141
10.4.4 Les clés ne sont pas nécessairement des chaînes de caractères 142
10.4.5 Les dictionnaires ne sont pas des séquences 143
10.4.6 Construction d'un histogramme à l'aide d'un dictionnaire 144
Chapitre 11 : Classes, objets, attributs 145
11.1 Utilité des classes 145
11.2 Définition d'une classe élémentaire 146
11.3 Attributs (ou variables) d'instance 147
11.4 Passage d'objets comme arguments lors de l'appel d'une fonction 148
11.5 Similitude et unicité 148
11.6 Objets composés d'objets 149
11.7 Objets comme valeurs de retour d'une fonction 150
11.8 Les objets sont modifiables 150
Chapitre 12 : Classes, méthodes, héritage 151
12.1 Définition d'une méthode 151
12.2 La méthode "constructeur" 153
12.3 Espaces de noms des classes et instances 157
12.4 Héritage 158
12.5 Héritage et polymorphisme 159
12.6 Modules contenant des bibliothèques de classes 163
Chapitre 13 : Classes & Interfaces graphiques 166
13.1 "Code des couleurs" : un petit projet bien encapsulé 166
13.2 "OscilloGraphe" : un widget personnalisé construit par dérivation 170
13.3 "Curseurs" : un widget composite 175
13.3.1 Présentation du widget "Scale" 175
13.3.2 Construction d'un panneau de contrôle à trois curseurs 176
13.4 Intégration de widgets composites dans une application synthèse 180
Chapitre 14 : Et pour quelques widgets de plus ... .184
14.1 Les "boutons radio" 184
14.2 Utilisation des cadres (frames) pour la composition d'une fenêtre 186
14.3 Comment déplacer des dessins à l'aide de la souris 188
14.4 Python Mega Widgets 191
14.4.1 "Combo Box" 191
14.4.2 "Scrolled Text" 192
14.4.3 "Scrolled Canvas" 195
14.4.4 Barres d'outils avec bulles d'aide - expressions lambda 198
14.5 Fenêtres avec menus 200
14.5.1 Première ébauche du programme : 201
14.5.2 Ajout de la rubrique "Musiciens" 203
14.5.3 Ajout de la rubrique "Peintres" : 205
14.5.4 Ajout de la rubrique "Options" : 206
14.6 Exemple de programme concret : jeu de Ping 211

14.6.1 Principe 211
14.6.2 Programmation 212
Chapitre 15 : CGI et Python 217
15.1 Pages Web interactives 217
15.2 L'interface CGI 218
15.3 Une interaction CGI rudimentaire 218
15.4 Un formulaire HTML pour l'acquisition des données 220
15.5 Un script CGI pour le traitement des données 221
Chapitre 16 : Gestion d'une base de données 223
16.1 Les bases de données 223
16.1.1 SGBDR - Le modèle client/serveur 223
16.1.2 Le langage SQL - Gadfly 224
16.2 Mise en oeuvre d'une base de données simple avec Gadfly 225
16.2.1 Création de la base de données 225
16.2.2 Connexion à une base de données existante 226
16.2.3 Recherches dans une base de données 227
16.2.4 La requête select 229
Chapitre 17 : Communications à travers un réseau 230
17.1 Les sockets 230
17.2 Construction d'un serveur élémentaire 231
17.3 Construction d'un client rudimentaire 233
Annexes 234
0.1 Solutions de quelques exercices 234
0.2 Quelques idées de projets à développer : 272
0.3 Annexes tirées du texte original "How to think like a computer scientist" 273
0.3.1 Preface 273
0.3.2 Contributor list 276
0.3.3 GNU Free Documentation License 278